AN EFFICIENT, EXTENSIBLE, HARDWARE-AWARE INDEXING KERNEL

by

Mohammad Sadoghi Hamedani

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

© Copyright 2013 by Mohammad Sadoghi Hamedani
Modern hardware has the potential to play a central role in scalable data management systems. A realization of this potential arises in the context of indexing queries, a recurring theme in real-time data analytics, targeted advertising, algorithmic trading, and data-centric workflows, and of indexing data, a challenge in multi-version analytical query processing. To enhance query and data indexing, in this thesis, we present an efficient, extensible, and hardware-aware indexing kernel. This indexing kernel rests upon novel data structures and (parallel) algorithms that utilize the capabilities offered by modern hardware, especially abundance of main memory, multi-core architectures, hardware accelerators, and solid state drives.

This thesis focuses on presenting our query indexing techniques to cope with processing queries in data-intensive applications that are susceptible to ever increasing data volume and velocity. At the core of our query indexing kernel lies the BE-Tree family of memory-resident indexing structures that scales by overcoming the curse of dimensionality through a novel two-phase space-cutting technique, an effective top-k processing, and adaptive parallel algorithms to operate directly on compressed data (that exploits the multi-core architecture). Furthermore, we achieve line-rate processing by harnessing the unprecedented degrees of parallelism and pipelining only available through low-level logic design using FPGAs. Finally, we present a comprehensive evaluation that establishes the superiority of BE-Tree in comparison with state-of-the-art algorithms.

In this thesis, we further expand the scope of our indexing kernel and describe how to accelerate analytical queries on (multi-version) databases by enabling indexes on the most recent data. Our goal is to reduce the overhead of index maintenance, so that indexes can be used effectively for analytical queries without being a heavy burden on transaction throughput. To achieve this end, we re-design the data structures in the storage hierarchy to employ an extra level of indirection over solid state drives. This indirection layer dramatically reduces the amount of magnetic disk I/Os that is needed for
updating indexes and localizes the index maintenance. As a result, by rethinking how data is indexed, we eliminate the dilemma between update vs. query performance and reduce index maintenance and query processing cost substantially.
Dedication

To my mother, Lili Taghavi.
Acknowledgements

“In God’s Great Name”

First and foremost, this thesis would have not been possible without the caring support and the ongoing and unlimited love of my mother, Lili Taghavi, who has placed my life above hers. This thesis is the fruit of continuous dedication and kindness of my mother. I will forever be grateful, yet regrettably incapable of emanating such a sincere love even to fill a small corner of my debt to her.

The flourishing of this thesis owes a great deal to my advisor, Professor Hans-Arno Jacobsen. His care and guidance have touched every aspect of my academic research horizon: how to formulate a problem, how to tackle a challenging problem, and how to express and to convey one’s ideas, to name a few. With patience and constant support and encouragement, he has nurtured my ability to undertake an academic research career.

In addition to my advisor, I had many great mentors at University of Toronto, McGill University, and IBM T.J. Watson that played a pivotal role in my academic development including Dr. R. Hull, Prof. K. Ross, Dr. Y-C. Chang, Prof. F. Bacchus, Prof. R. Johnson, Prof. B. Kemme, Prof. P. Andritsos, Prof. A. D. Brown, Prof. R. Miller, and Prof. N. Koudas. I also had the pleasure of working with unique colleagues at IBM T.J. Watson, notably, B. Bhattacharjee, M. Canim, O. Hassanzadeh, V. Muthusamy, and R. Vaculin.

My life is blessed with a warm and loving family, a life that will be void without their presence. From early childhood, the unconditional love of my grandmother and my grandfather, my cousins, my aunts, and my uncles have been vital in overcoming all hardships in life and attaining my happiness and success.

My journey has been full of wonderful friends that without a doubt fills a special place in my life. I am grateful to my long lasting friendship that were formed in high school (Ali and Behnam); to the true friendship that were unfolded in undergraduate period (Ivona, Kristen, Mahendan, Mario, Matt, Nafis, and Will); and, finally, to many great friends that were a big part of my amazing graduate school endeavor (Albert, Amer, Chen, Dimitris, Fatemeh, Fei, Golnaz, Harsh, Jiang, Kianoosh, Kaiwen, Maniy, Manos, Michael, Nikos, Nilesh, Oktie, Patricia, Reza, Sara, Shirin, Tilmann, Vinod, and Young).
# Contents

## I Overview

### 1 Introduction

1.1 Motivation .................................................. 2
1.2 Thesis Problem Statement .................................. 6
1.3 Thesis Contributions ....................................... 8
   1.3.1 Query Indexing ........................................ 8
   1.3.2 Parallel Query Indexing ............................... 8
   1.3.3 Top-k Query Indexing .................................. 8
   1.3.4 Data Indexing .......................................... 9
   1.3.5 XML/XPath-to-Kernel Mapping ......................... 9
   1.3.6 Data-centric Workflows-to-Kernel Mapping .......... 9
   1.3.7 FPGA Query Indexing .................................. 10
   1.3.8 FPGA SQL Query Indexing .............................. 10
1.4 Thesis Organization ...................................... 10

## 2 Related Work

2.1 Query Indexing ............................................. 12
2.2 Parallel Query Indexing .................................. 14
2.3 Top-k Query Indexing ..................................... 15
2.4 Data Indexing and SSDs ................................... 16
2.5 XML/XPath Processing ..................................... 17
2.6 FPGAs Acceleration ....................................... 19
2.7 Data-centric Workflow Execution ........................ 20

## II Kernel Core

### 3 Query Indexing (Main Memory)

3.1 Introduction ............................................... 23
3.2 Expression Matching Model ............................... 26
   3.2.1 Expression Language ................................ 26
   3.2.2 Matching Semantics .................................. 26
3.3 BE-Tree Organization ..................................... 28
   3.3.1 BE-Tree Structure ................................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>93</td>
</tr>
<tr>
<td>5</td>
<td>Top-k Query Indexing (Main Memory)</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>94</td>
</tr>
<tr>
<td>5.2</td>
<td>Language and Data Model Formalism</td>
<td>96</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Notation</td>
<td>96</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Expression Language and Subspace Model</td>
<td>96</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Top-k Matching Semantics</td>
<td>97</td>
</tr>
<tr>
<td>5.3</td>
<td>BE*-Tree Structural Properties</td>
<td>98</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Non-rigid Space-cutting</td>
<td>99</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Structural Adaptation</td>
<td>100</td>
</tr>
<tr>
<td>5.3.3</td>
<td>BE*-Tree Invariance</td>
<td>101</td>
</tr>
<tr>
<td>5.4</td>
<td>BE*-Tree Adaptiveness Properties</td>
<td>102</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Bi-directional Expansions</td>
<td>102</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Adaptive Ranking Function</td>
<td>108</td>
</tr>
<tr>
<td>5.5</td>
<td>Top-k Algorithm</td>
<td>109</td>
</tr>
<tr>
<td>5.6</td>
<td>Evaluations</td>
<td>111</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Experiment Overview</td>
<td>111</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Micro Experiments</td>
<td>112</td>
</tr>
<tr>
<td>5.6.3</td>
<td>Macro Experiments</td>
<td>113</td>
</tr>
<tr>
<td>5.7</td>
<td>Summary</td>
<td>118</td>
</tr>
<tr>
<td>6</td>
<td>Data Indexing (SSD)</td>
<td>120</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>120</td>
</tr>
<tr>
<td>6.2</td>
<td>Basic Indirection Structure</td>
<td>122</td>
</tr>
<tr>
<td>6.3</td>
<td>Enhancing Insertions</td>
<td>124</td>
</tr>
<tr>
<td>6.3.1</td>
<td>LIDBlocks</td>
<td>124</td>
</tr>
<tr>
<td>6.3.2</td>
<td>The Dense Index Case</td>
<td>125</td>
</tr>
<tr>
<td>6.3.3</td>
<td>The Sparse Index Case</td>
<td>126</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Revisiting Deletions and Updates</td>
<td>126</td>
</tr>
<tr>
<td>6.3.5</td>
<td>DeltaBlock Technique</td>
<td>126</td>
</tr>
<tr>
<td>6.4</td>
<td>Extended Example</td>
<td>128</td>
</tr>
<tr>
<td>6.5</td>
<td>Analysis</td>
<td>131</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Time Complexity</td>
<td>131</td>
</tr>
<tr>
<td>6.5.2</td>
<td>SSD Space Complexity</td>
<td>131</td>
</tr>
<tr>
<td>6.5.3</td>
<td>SSD Life-Expectancy</td>
<td>131</td>
</tr>
<tr>
<td>6.6</td>
<td>Evaluations</td>
<td>132</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Platform</td>
<td>132</td>
</tr>
<tr>
<td>6.6.2</td>
<td>DB2 Query Re-writing Experiments</td>
<td>133</td>
</tr>
<tr>
<td>6.6.3</td>
<td>GiST Implementation</td>
<td>137</td>
</tr>
<tr>
<td>6.6.4</td>
<td>DB2 Operational Data Store Experiments</td>
<td>140</td>
</tr>
<tr>
<td>6.7</td>
<td>Summary</td>
<td>142</td>
</tr>
</tbody>
</table>
III Kernel Extensions

7 XML/XPath-to-Kernel Mapping

- 7.1 Introduction .................................................. 145
- 7.2 XPath Expression Encoding ............................... 145
- 7.3 XPath Encodings ............................................. 148
  - 7.3.1 XPath Expressions .................................... 148
  - 7.3.2 XPath encoding illustration ...................... 152
  - 7.3.3 XPath encoding dichotomy ...................... 152
- 7.4 XML Document Encoding ................................. 153
  - 7.4.1 XML encoding illustration ....................... 156
- 7.5 XML/XPath Matching ....................................... 156
- 7.6 Summary ..................................................... 157

8 Data-centric Workflows-to-Kernel Mapping

- 8.1 Introduction ................................................. 158
- 8.2 Data-centric Workflow .................................... 160
  - 8.2.1 Overview of GSM Schema ......................... 160
  - 8.2.2 Generalized Data-centric Workflow ............. 161
- 8.3 Publish/Subscribe Schema ............................... 162
- 8.4 Workflow Mapping Overview ............................ 166
- 8.5 Mapping Formalization .................................... 167
  - 8.5.1 Matching and Notification Policies ............. 167
  - 8.5.2 Consumption Policy ................................. 172
- 8.6 Workflow Mapping Analysis ............................. 173
- 8.7 Foundation of Distribution .............................. 177
- 8.8 Summary ..................................................... 179

IV Kernel on Reconfigurable Hardware

9 FPGA Query Indexing

- 9.1 Introduction .................................................. 182
- 9.2 Event Processing Model .................................. 184
- 9.3 FPGA-based Event Processing ........................... 184
  - 9.3.1 Tuning for Flexibility .............................. 185
  - 9.3.2 Tuning for Adaptability ............................ 186
  - 9.3.3 Tuning for Scalability ............................. 188
  - 9.3.4 Tuning for Performance ............................ 189
- 9.4 Evaluations .................................................. 189
  - 9.4.1 Platform & Evaluation Setting .................... 190
  - 9.4.2 Evaluation Results ................................. 191
- 9.5 Summary ..................................................... 192
10 FPGA SQL Query Indexing
10.1 Introduction .......................................................... 194
10.2 FPGA Event Stream Processing Model ......................... 195
10.3 Operator-to-Circuit Mappings .................................... 195
10.4 Multi-query-to-Circuit Mappings ................................. 200
10.5 Platform Architecture and Setup ................................. 201
10.6 Summary .................................................................. 202

11 Conclusions ............................................................... 203
11.1 Summary of Results ..................................................... 203
11.2 Future Work ............................................................. 204

Bibliography ................................................................. 208
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Operator Transformation</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Levels of Expressiveness</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>Inferred Predicates from $P^{(i,v_*)}$</td>
<td>51</td>
</tr>
<tr>
<td>3.4</td>
<td>Experiment Settings for Synthetic Datasets</td>
<td>52</td>
</tr>
<tr>
<td>3.5</td>
<td>Experiment Settings for Real Datasets</td>
<td>52</td>
</tr>
<tr>
<td>3.6</td>
<td>Most Effective $\text{max}_{cap}$ for Different Matching Probabilities</td>
<td>55</td>
</tr>
<tr>
<td>3.7</td>
<td>BE-Tree Construction Time &amp; Index Size</td>
<td>56</td>
</tr>
<tr>
<td>3.8</td>
<td>BE-Tree/Bitmap Construction Time (second) &amp; Memory Usage (MB)</td>
<td>69</td>
</tr>
<tr>
<td>3.9</td>
<td>Comparing BE-Tree (PC) and CLCB (GPU)</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Synthetic and Real Workload Properties</td>
<td>86</td>
</tr>
<tr>
<td>4.2</td>
<td>A-PCM matching time breakdown (%)</td>
<td>88</td>
</tr>
<tr>
<td>5.1</td>
<td>Predicate Mapping</td>
<td>97</td>
</tr>
<tr>
<td>5.2</td>
<td>Synthetic Workload Configuration</td>
<td>112</td>
</tr>
<tr>
<td>5.3</td>
<td>Real Workload Configuration</td>
<td>112</td>
</tr>
<tr>
<td>5.4</td>
<td>BE*-Tree Micro Experiments</td>
<td>112</td>
</tr>
<tr>
<td>6.1</td>
<td>Base vs. Indirection technique analysis</td>
<td>130</td>
</tr>
<tr>
<td>6.2</td>
<td>Query re-writing to capture indirection mechanism</td>
<td>133</td>
</tr>
<tr>
<td>9.1</td>
<td>Latency ($\mu$s) vs. the # of MUs (Scalability Design)</td>
<td>191</td>
</tr>
<tr>
<td>9.2</td>
<td>End-to-end System Latency ($\mu$s)</td>
<td>192</td>
</tr>
<tr>
<td>9.3</td>
<td>System Throughput (market events/sec)</td>
<td>192</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Targeted Advertising .................................................. 3
1.2 Thesis Overview (Indexing Kernel) .............................. 6

2.1 Top-k Structural Comparisons .................................... 16

3.1 The BE-Tree data structure. ........................................ 30
3.2 The BE-Tree directories. ............................................ 31
3.3 The BE-Tree clustering directories. ............................... 35
3.4 A Concrete Example ................................................ 40
3.5 BE-Tree Lazy Predicate Evaluation Technique .................. 46
3.6 BE-Tree Bitmap-based Predicate Evaluation Technique ...... 48
3.7 Matching Probability with Different $l$-node Capacities .......... 54
3.8 Extended Matching Probability with Different $l$-node Capacities .... 55
3.9 Matching Probability with Different Scoring Functions .......... 56
3.10 Matching Probability with Different Clustering Directory Types .. 57
3.11 Effect of Workload Size .......................................... 58
3.12 Effect of Dimensionality ......................................... 58
3.13 Effects of Dimension Cardinality ............................... 59
3.14 Effect of Dimension Selectivity .................................. 59
3.15 Effect of Predicate Selectivity ................................. 60
3.16 Effect of Subscription/Event Size ............................... 61
3.17 Effect of Percentage of Equality Predicates .................... 62
3.18 Effect of Percentage of Matching Probability .................. 63
3.19 Effect of Percentage of Matching Probability (DBLP) ......... 64
3.20 Effect of Percentage of Extended Matching Probability ........ 65
3.21 Effect of Real Workload Size .................................... 66
3.22 Effect of Events Expressiveness .................................. 67
3.23 Effect of Dynamic Workload .................................... 68
3.24 Effect of Dynamic Workload (Predicate False Candidate Rate) .. 68
3.25 Effects of Dimensionality/Dimension Cardinality on Lazy/Bitmap Optimizations .. 69
3.26 Effect of Percentage of Matching Probability on Lazy/Bitmap Optimizations ... 70
3.27 Effect of Sub/Event Size on Lazy/Bitmap Optimizations ....... 71
3.28 Effect of Cache-conscious Data Structure on Bitmap Optimization .......... 71
3.29 Effect of Workload Size on Bloom Filter Optimization .......... 72
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 BE-Tree Bitmap-based Event Encoding Technique</td>
<td>77</td>
</tr>
<tr>
<td>4.2 Convex vs. non-convex minimum bounding area</td>
<td>79</td>
</tr>
<tr>
<td>4.3 Overview of Matching Algorithm using BE-Tree Bitmap-based Encoding</td>
<td>79</td>
</tr>
<tr>
<td>4.4 Overview of BE-Tree Parallel/Matching on Compressed Events</td>
<td>80</td>
</tr>
<tr>
<td>4.5 BE-Tree Re-organization for Parallel Compressed Events Matching</td>
<td>81</td>
</tr>
<tr>
<td>(Stages 4-5 in Figure 4.4)</td>
<td></td>
</tr>
<tr>
<td>4.6 Overview of Stream Online Re-ordering and Matching</td>
<td>84</td>
</tr>
<tr>
<td>4.7 Comparison of various compression algorithm extensions</td>
<td>87</td>
</tr>
<tr>
<td>4.8 Effects of the compressed matching internal parameters</td>
<td>88</td>
</tr>
<tr>
<td>4.9 Comparison of sequential, parallel, and adaptive parallel compressed techniques</td>
<td>89</td>
</tr>
<tr>
<td>4.10 Comparison of sequential, parallel, and adaptive parallel compressed techniques</td>
<td>91</td>
</tr>
<tr>
<td>4.11 Latency comparison of sequential (Bitmap) and adaptive parallel compressed techniques (A-PCM) with different batch sizes from 128 to 1024 events (for brevity, we omit the name A-PCM in the legend, and we distinguish among different A-PCM run based on the batch sizes)</td>
<td>91</td>
</tr>
<tr>
<td>4.12 Low-level CPU performance counter analysis</td>
<td>92</td>
</tr>
<tr>
<td>5.1 The BE*-Tree data structure</td>
<td>99</td>
</tr>
<tr>
<td>5.2 Top-down vs. Bottom-up Expansion</td>
<td>102</td>
</tr>
<tr>
<td>5.3 Bucket Splitting</td>
<td>103</td>
</tr>
<tr>
<td>5.4 BE*-Tree Top-k Structural Overview</td>
<td>109</td>
</tr>
<tr>
<td>5.5 Varying Workload Size</td>
<td>113</td>
</tr>
<tr>
<td>5.6 Varying Space Dimensionality</td>
<td>114</td>
</tr>
<tr>
<td>5.7 Varying Subscription/Event Size</td>
<td>115</td>
</tr>
<tr>
<td>5.8 Varying Cardinality</td>
<td>115</td>
</tr>
<tr>
<td>5.9 Varying the Number of Clusters &amp; Cluster Size</td>
<td>116</td>
</tr>
<tr>
<td>5.10 Varying Matching Probability</td>
<td>117</td>
</tr>
<tr>
<td>5.11 Top-k vs. Workload Size (Match Prob 0.1%)</td>
<td>118</td>
</tr>
<tr>
<td>5.12 Top-k vs. Matching Probability</td>
<td>119</td>
</tr>
<tr>
<td>6.1 Effect of Adding Indexes in Operational Data Store</td>
<td>121</td>
</tr>
<tr>
<td>6.2 Traditional RID index structure</td>
<td>123</td>
</tr>
<tr>
<td>6.3 LID index using the indirection technique</td>
<td>124</td>
</tr>
<tr>
<td>6.4 Indirection Technique with LIDBlock</td>
<td>125</td>
</tr>
<tr>
<td>6.5 Indirection Technique with DeltaBlock</td>
<td>127</td>
</tr>
<tr>
<td>6.6 Varying the number of indexes vs. update/query time</td>
<td>134</td>
</tr>
<tr>
<td>6.7 Varying the number of indexes vs. page writes</td>
<td>135</td>
</tr>
<tr>
<td>6.8 Varying the query selectivity vs. query execution time</td>
<td>136</td>
</tr>
<tr>
<td>6.9 Varying the indirection implementation techniques</td>
<td>136</td>
</tr>
<tr>
<td>6.10 Insert/Update/Query execution time</td>
<td>137</td>
</tr>
<tr>
<td>6.11 Varying the number of indexes vs. update time</td>
<td>138</td>
</tr>
<tr>
<td>6.12 Varying the bufferpool size vs. update time</td>
<td>139</td>
</tr>
<tr>
<td>6.13 Varying the LIDBlock size vs. insertion time</td>
<td>140</td>
</tr>
<tr>
<td>6.14 Varying the LIDBlock size vs. insertion time</td>
<td>140</td>
</tr>
<tr>
<td>6.15 Varying the DeltaBlock size vs. update time</td>
<td>141</td>
</tr>
</tbody>
</table>
List of Algorithms

1 MatchBETree \((event, cnode, matchedSub)\) ........................................ 35
2 SearchCDir \((event, cdir, matchedSub)\) .......................................... 36
3 InsertBETree \((sub, cnode, cdir)\) ....................................................... 37
4 InsertCDir \((sub, cdir)\) ................................................................. 37
5 SpacePartitioning \((cnode)\) .............................................................. 38
6 SpaceClustering \((cdir)\) ................................................................. 38
7 DeleteBETree \((sub, cnode)\) ............................................................. 39
8 SearchDeleteCDir \((sub, cdir)\) ......................................................... 39
9 Insert \((\Omega, cnode, cdir)\) ........................................................... 104
10 InsertCDir \((\Omega, cdir)\) ............................................................... 105
11 UpdateBucketInfo \((\Omega, cdir.B^k)\) ............................................. 105
12 Partitioning \((cnode)\) ................................................................. 106
13 Clustering \((cdir)\) ................................................................. 106
14 MatchTopk \((\omega, \Omega_{matched}, cnode)\) ..................................... 110
15 SearchCDir \((\omega, \Omega_{matched}, cdir)\) ....................................... 110
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BE-Tree</td>
<td>Boolean Expression-Tree</td>
</tr>
<tr>
<td>BE*-Tree</td>
<td>Boolean Expression-Tree with Top-k Processing</td>
</tr>
<tr>
<td>PCM</td>
<td>Parallel Compressed Matching Algorithm</td>
</tr>
<tr>
<td>A-PCM</td>
<td>Adaptive Parallel Compressed Matching Algorithm</td>
</tr>
<tr>
<td>OSR</td>
<td>Online Stream Re-ordering Algorithm</td>
</tr>
<tr>
<td>BEGen</td>
<td>Boolean Expression Workload Generator</td>
</tr>
<tr>
<td>GPX-Matcher</td>
<td>Generic Boolean Predicate-based XPath Expression Matcher</td>
</tr>
<tr>
<td>Pub/Sub</td>
<td>Publish/Subscribe</td>
</tr>
<tr>
<td>GSM</td>
<td>Guard-Stage-Milestone Data-centric Workflow</td>
</tr>
<tr>
<td>B-step</td>
<td>Business-step in Data-centric Workflows</td>
</tr>
<tr>
<td>SQL</td>
<td>Standard Query Language</td>
</tr>
<tr>
<td>PSQL</td>
<td>PADRES Standard Query Language</td>
</tr>
<tr>
<td>SPJ</td>
<td>Select-Project-Join Query</td>
</tr>
<tr>
<td>OLTP</td>
<td>Online Transaction Processing</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-programmable Gate Array</td>
</tr>
<tr>
<td>FQL</td>
<td>FPGA Query Language</td>
</tr>
<tr>
<td>fpga-ToPSS</td>
<td>fpga-Toronto Publish/Subscribe System</td>
</tr>
<tr>
<td>SSD</td>
<td>Solid-State Drive</td>
</tr>
</tbody>
</table>
Part I

Overview
Chapter 1

Introduction

1.1 Motivation

Query Indexing Kernel

The efficient event processing (query indexing) of large collections of patterns (expressed as Boolean expressions, XPath expressions, or continuous SQL queries) over event streams plays a central role in major data-intensive applications ranging from user-centric processing and personalization to real-time data analysis. On the one hand, emerging user-centric applications, including targeted advertising and selective information dissemination, demand determining and presenting to an end-user only the most relevant content that is both user-consumable and suitable for limited screen real estate of target (mobile) devices. We achieve these user-centric requirements through novel high-dimensional indexing structures and (parallel) algorithms. On the other hand, applications in real-time data analysis, including algorithmic trading, data-centric workflow execution, and intrusion detection, demand meeting stringent subsecond processing requirements and providing high-frequency and low-latency event processing over event streams. We achieve real-time data analysis requirements by leveraging multi-core architectures and reconfigurable hardware – field-programmable gate arrays (FPGAs). In order to sustain line-rate processing, we leverage shared-memory architecture prevalent in multi-core processors, and we exploit unprecedented degrees of parallelism and potential for pipelining, only available through custom-built, application-specific, and low-level logic design. Finally, we conduct a comprehensive evaluation to demonstrate the superiority of our proposed techniques in comparison with state-of-the-art algorithms designed for event processing.

Efficient event processing is an integral part of a growing number of web and data management technologies ranging from user-centric processing and personalization to real-time data analysis. In user-centric processing applications, there are targeted advertising [211, 82], online job sites [140, 211], and location-based services for emerging applications in the co-spaces [11, 160]; common to all are patterns and specifications (e.g., advertising campaigns, job profiles, and service descriptions) modeled as Boolean expressions, XPath expressions, or SQL queries and incoming user information (e.g., user profiles and preferences) modeled as events using attribute-value pairs, XML document, or relational tuples. In the real-time analysis domain, there are (complex) event processing [83, 12, 38, 74, 36], XML filtering [14, 67, 148, 131, 132], intrusion detection [196, 77], data-centric workflow execution [64, 108], and algorithmic trading [182]; again, common among these applications are predefined set of patterns.
(e.g., investment strategies and attack specifications) modeled as subscriptions and streams of incoming data (e.g., XML documents, data packets, and stock feeds) modeled as events.

The prominent example of user-centric processing is the targeted advertising scenario, which is shown in Figure 1.1. In this scenario, incoming users are modeled as an event stream and advertisers’ campaigns are modeled as a collections of subscriptions. The role of indexing kernel is to efficiently determine (most) relevant ads for each user.

Unique to user-centric processing and personalization are strict requirements to determine only the most relevant content (e.g., ads) that is both user-consumable and suitable for the often limited screen real estate of client devices [140, 211, 82]. In addition, the user-centric processing demands scaling to millions of patterns and specifications (e.g., advertising campaigns) for supporting large-scale enterprise-level user-services, processing latency constraints in the subsecond range for meeting an acceptable service-level agreement, and improve expression expressiveness for capturing interesting patterns and desired preferences. To address these challenges using a software-based approach, we develop and design an effective algorithms and high-dimensional indexing structures [175, 77, 176, 178] to achieve processing large volume of incoming user information and to serve user-relevant contents.

Unique to real-time data analysis applications are critical requirements to meet the ever growing demands in processing large volumes of data at predictably low-latencies across many application scenarios [196, 144, 103, 55, 109, 63]. The need for more processing bandwidth is the key ingredient in high-throughput real-time data analysis that enables processing, analyzing, and extracting relevant information from streams of incoming data. Therefore, as proliferation of data and bandwidth continues, it is becoming essential to expand the research horizon to go beyond the conventional software-based approaches and adopt other key enabling technologies such as reconfigurable hardware in the form of FPGAs. FPGAs are cost-effective and energy-efficient solutions that are increasingly being explored.
to accelerate data management applications. On this front, using a hardware-based approach, through FPGAs, we exploit the inherent hardware parallelism by creating custom-built, application-specific, and low-level logic design in order to achieve real-time event processing on hardware [182, 183, 182, 181].

In particular, at the core of any efficient event processing platform lies a query indexing kernel that solves the matching problem (i.e., the key focus of this thesis). Informally, we define the stateless matching problem\(^1\) as:

**Definition 1.** Given an event \(\omega\) (e.g., a user profile) and a set of Boolean expressions \(\Omega\) (e.g., advertising campaigns), find all expressions \(\Omega_i \in \Omega\) satisfied by \(\omega\).

This definition can be extended to support stateful matching (i.e., complex event processing) by incorporating time- or count-based sliding window semantics in order to re-define the matching problem over a snapshot of an observed finite portion of the event stream. In addition, our expressions are reformulated as SQL queries (instead of Boolean expressions) extended with sliding window semantics. Likewise, the stateful matching problem is defined as follows:

**Definition 2.** Given a stream of events (i.e., a bag of \(\langle \omega, \tau \rangle\) tuples, where \(\omega\) is an event at time \(\tau\)) and a collection of continuous SQL queries \(Q\), the SQL queries are continuously evaluated over the event stream.

We extract several challenges of paramount importance from applications that rely on event processing, in particular, when patterns are defined as Boolean expressions, the primary focus of this thesis. First and foremost, the index structure designed for matching Boolean expressions must support top-k matching to quickly retrieve only the most relevant expressions, which is not addressed by prior Boolean expression matching approaches (e.g., [12, 74]). The relevance computation must be based on a generic and preference-aware scoring function, which is not addressed by prior approaches (e.g., [140]). Furthermore, the top-k model must cope with much higher dimensionality (e.g., a dimension can represent an attribute in the user profile such as age, gender, and location) that is beyond the scope of dominant database top-k techniques (e.g., [112]) and skyline query processing (e.g., [34]). In general, the prevalent space dimensionality in event processing applications is in the hundreds (or thousands), namely, orders of magnitude larger than capabilities of existing multi-dimensional and high-dimensional structures developed in the database community (e.g., [87]). Second, the index must support predicates with an expressive set of operators over continuous and discrete domains, which is also not supported by prior matching approaches (e.g., [12, 74, 211]). Third, the index must enable dynamic insertion and deletion of expressions, often disregarded as a requirement in matching algorithm design (e.g., [12, 140, 211]). Fourth, the index must employ a dynamic structure that adapts to changing workload distributions and expression schemata, also often disregarded in most matching algorithm designs (e.g., [12, 140, 211]).

Moreover, we argue that an index structure must scale to millions of Boolean expressions and afford low-latency and high-throughput expression matching demanded by most user-centric and real-time data analysis applications; thus, sixth, the index must exploit emerging parallel hardware, which have received little attention by prior Boolean expression matching approaches (e.g., [12, 74, 140, 211, 82, 176, 178]). Seventh, the index must be stream-aware in order to identify and utilize the overlap (or the similarity) within the event stream. Again such fresh perspectives are largely ignored by the prior-art in matching algorithms (e.g., [12, 74, 140, 211, 82, 176, 178]). Finally, we claim that in order to scale to the internet-

\(^1\)Various semantics of stateless matching problem are defined in Section 3.2
scale volume of content that is served to millions of users in user-centric applications such as social networks, any efficient event processing must scale to many processing nodes in a distributed setting. Therefore, to facilitate users to sift through the data for relevant information and avoid overwhelming the user, considering the vast amount of content on social networks, eight, the index must support distributed top-k processing, which is not addressed by earlier work (e.g., \[140, 211, 82, 178\]).

In what follows, we use the terms “event stream” and “stream” interchangeably. Similarly, the terms “query indexing”, “(complex) event processing” and “(complex) event matching” are used interchangeably. This flexibility in terminology is also reflected in related literature that spans the database, the distributed and parallel computing, the event-based computing, and the middleware communities. We formulate our “event matching problem” using the well-known publish/subscribe (pub/sub) matching problem \[12, 74, 140, 211, 76, 82, 176, 178\] described in Section 3.2.

### Data Indexing Kernel

The second aspect of our indexing kernel, is how to efficiently index multi-version databases in order to improve analytical query processing. In a multi-version (e.g., bi-temporal) database system, new records do not physically replace old ones. Instead, a new version of the record is created, which becomes visible to other transactions at commit time. Conceptually, there may be many versions of a record, each corresponding to the state of the database at some point in the past (an indispensable property for conforming to auditing and compliances regulations). However, when indexing data to enable a typical analytical query workload, one tends to index only the most recent version of the data since the latest version is most commonly accessed \[53, 162, 110, 193\]. In such a setting, record insertions, deletions, and updates trigger expensive physical disk I/Os to keep the indexes up-to-date. With a traditional index structure, for example, updating a record triggers a traversal of every single index to point to the new version of the record. These overheads have historically been problematic for online transaction processing (OLTP) workloads that are update-intensive and, thus, resulted in reducing the number of available indexes. This choice substantially hinders the efficiency of executing analytical queries, which are often important for operational data stores.

Therefore, the objective of a data indexing kernel is to reduce the overhead of index maintenance, so that indexes can be used effectively for analytical queries without being a heavy burden on transaction throughput. To fulfill this objective, we re-design the data structures in the storage hierarchy to employ an extra level of indirection through a solid state storage (SSD) layer, unlike previous approaches \[40, 121, 31, 41, 68\], which simply stored “hot” data (i.e., most recently accessed data) on SSDs. Because the solid state memory is fast (orders of magnitude faster than disks for small random I/Os), the extra time incurred by the extra level of indirection is small. This extra level of indirection dramatically reduces the amount of magnetic disk I/Os that is needed for index updates and localizes the maintenance to only indexes that are defined on the updated columns as opposed to all indexes. In addition, by extending our indirection layer, we are able to batch updates (and inserts) in order to further reduce physical disk I/Os and disk usage (through compression).

---

\(^{2}\)As of October 2012, Facebook has over a billion active users; \[http://newsroom.fb.com/content/default.aspx?NewsAreaId=22\].
1.2 Thesis Problem Statement

In this thesis, we develop an efficient, extensible, hardware-aware indexing kernel. This indexing kernel revolves around three distinguishing properties: efficiency, which is achieved by proposing novel data structures and (parallel) algorithms; extensibility, which is achieved by expanding the applicability of our indexing kernel to XML filtering and workflow execution; and hardware-awareness, which is achieved by employing reconfigurable hardware, multi-cores, and SSDs. In Figure 1.2, we illustrate the various components of our indexing kernel.\(^3\)

The main building blocks of our solution consists of an efficient query indexing kernel\(^4\) (cf. Chapters 3-5) and data indexing kernel (cf. Chapter 6). The efficiency of our query indexing kernel\(^4\) is partly addressed through a novel high-dimensional indexing structure [176, 179], top-k matching [178], parallel compressed matching, distributed top-k matching [221], and (complex) event processing [77, 170]. Furthermore, the extensibility of this query indexing kernel paves the way towards solving XML filtering and dissemination [175] and data-centric workflow execution (cf. Chapters 7-8).

Another major benefit of our indexing kernel is its regular, more tabular organization of its data structure, as well as Boolean expressions and attribute-value pairs-based nature of its base language, that lend itself well to parallel and hardware processing. As a result, by tapping into the underlying hardware, our indexing kernel is further boosted by leveraging an unprecedented hardware parallelism [180, 182, 183, 181], only available through custom-built, application-specific, and low-level logic design (cf. Chapter 9-10).

At the core of our query indexing kernel, we present BE-Tree (Boolean Expression-Tree) family structure that achieves scalability by overcoming the curse of dimensionality through a (non-rigid) space-

---

\(^3\)As part of this thesis, we also developed a comprehensive Boolean expression workload generator (BEGen): http://msrg.org/datasets/BEGen.

\(^4\)The query indexing kernel is implemented in C (≈ 50K lines of code) and data indexing kernel is implemented in C++ (totaling ≈ 8K lines of code).
Chapter 1. Introduction

cutting technique without restricting the language expressiveness and structure dynamics [176, 178, 179]. Notably, with respect to scalability, in BE*-Tree, we solve the two critical challenges common to most multi-dimensional and high-dimensional structures (e.g., [87]): (1) avoiding indexing empty space (2) minimizing overlap and coverage [178]. Furthermore, BE*-Tree is geared towards efficiently determining the most relevant expressions (e.g., ads), in which top-k processing is treated as a first class citizen [178]. BE*-Tree introduces a novel hierarchical top-k processing scheme that differs from existing work which relies on a static and a flat structure [140, 211]. These challenges are tackled in BE*-Tree by proposing a bi-directional tree expansion: (1) a top-down (data and space clustering) and a bottom-up (space clustering) growths process, which together enable indexing only non-empty continuous subspaces and adapting to workload changes; (2) a splitting strategy to systematically produce and maintain overlap-free subspaces for holding expressions [178]. Moreover, with respect to adaptability, in BE-Tree, we introduce a deterministic and a self-adjusting mechanism that adapts as expression and event workload changes; thus, BE-Tree’s main focus is to achieve an insertion-sequence independent dynamics [176].

Finally, within the core of our kernel, on the one hand, we enhance the query indexing by enabling parallel matching over multi-core shared-memory architectures and by supporting distributed top-k matching over shared-nothing architecture [221] (the distributed top-k matching is not included in this manuscript). On the other hand, with regards to data indexing, we develop an Indirection technique by expanding memory storage hierarchy with solid state drives (SSDs) to reduce index maintenance and database size that are crucial for improving the execution of analytical queries over mutl-vision (i.e., bi-temporal) and key-value stores.

To achieve high-throughput and low-latency indexing kernel, we reach out to hardware acceleration techniques on reconfigurable hardware in form of FPGAs [180, 182, 183, 181]. The hardware acceleration encompasses both classes of matching problems: stateless and stateful matching. The former problem is realized through novel custom-built, application-specific, and low-level logic designs that targets specific higher-level application needs such as flexibility, adaptability, scalability, and performance. The latter problem is fulfilled by enabling multi-query optimizations of SPJ (Select-Project-Join) queries inspired by highly parallelizable rule-based systems. The overlap among SPJ query plans is exploited by identifying unique intra- and inter-parallelism for window-based-join semantics and constructing a single global query plan to be executed on hardware.

To demonstrate the extensibility dimension of our indexing kernel, we reformulate the problem of XML filtering using our kernel. We develop a novel mapping of XPath expressions and XML documents into Boolean expressions and attribute-value pairs, respectively, that leverages our indexing kernel [175]. We present a dichotomy of the XPath language that identifies a subset of XPath language that can be mapped into our novel Boolean expression encoding. Moreover, in order to study the potential of our indexing kernel (as the engine of publish/subscribe systems), we also mapped data-centric workflows to the publish/subscribe abstraction. In essence, we present a novel reformulation of data-centric workflow that is designed to utilize the loosely coupled and distributed nature of the publish/subscribe abstraction.

\footnote{As part of our query indexing kernel, we also developed a novel Access Predicate Pruning (APP) index structure [77], which is evaluated in Section 3.8, but the structure details are not included in this manuscript.}

\footnote{Our batching technique using SSDs also improves insertion/deletion time for relational databases.}
1.3 Thesis Contributions

In this section, we outline the key contributions of this thesis.

1.3.1 Query Indexing

In Chapter 3, we present BE-Tree, a novel dynamic data structure designed to efficiently index Boolean expressions over a high-dimensional discrete space; parts of this chapter are published in [77, 176, 179, 174, 177]. BE-Tree copes with both high-dimensionality and expressiveness of Boolean expressions by introducing an effective two-phase space-cutting technique that specifically utilizes the discrete and finite domain properties of the space. Furthermore, BE-Tree employs self-adjustment policies to dynamically adapt the tree as the workload changes. Moreover, in BE-Tree, we develop two novel cache-conscious predicate evaluation techniques, namely, lazy and bitmap evaluations, that also exploit the underlying discrete and finite space to substantially reduce BE-Tree's matching time by up to 75%. We also introduce a Bloom filtering strategy to further improve search space pruning in BE-Tree.

BE-Tree is a general index structure for matching Boolean expression which has a wide range of applications including (complex) event processing, publish/subscribe matching, emerging applications in co-spaces, profile matching for targeted web advertising, and approximate string matching. Finally, the superiority of BE-Tree is proven through a comprehensive evaluation with state-of-the-art index structures designed for matching Boolean expressions.

1.3.2 Parallel Query Indexing

We argued that the efficient processing of large collections of patterns expressed as Boolean expressions over event streams plays a central role in major data-intensive applications ranging from user-centric processing and personalization to real-time data analysis. On the one hand, emerging user-centric applications, including targeted advertising and selective information dissemination, demand determining and presenting to an end-user the relevant content as it is published. On the other hand, applications in real-time data analysis, including algorithmic trading, and intrusion detection demand meeting stringent subsecond processing requirements and providing high-frequency event processing. We achieve these event processing requirements by exploiting the shift towards multi-core architectures by proposing a novel adaptive parallel compressed event matching algorithm (A-PCM) and an efficient online event stream re-ordering algorithm (OSR) that realize an unprecedented degree of parallelism, in Chapter 4; parts of this chapter are published in [174]. In our comprehensive evaluation, we demonstrate the superiority of our proposed techniques; we show our proposed adaptive parallel compressed event matching sustains an event rate of up to 233,863 events/second while state-of-the-art sequential event matching algorithm sustains only 36 events/second when processing up to five million Boolean expressions.

1.3.3 Top-k Query Indexing

Most notably, emerging user-centric applications such as targeted advertising and selective information dissemination demand determining and presenting to an end-user only the most relevant content that is both user-consumable and suitable for limited screen real estate of target devices. To retrieve the most relevant content, in Chapter 5, we present BE*-Tree, a novel indexing data structure designed for
effective hierarchical top-\(k\) pattern matching, which as its by-product also reduces the operational cost of processing millions of patterns; parts of this chapter are published in [178, 174].

To further reduce processing cost, BE*-Tree employs an adaptive and non-rigid space-cutting technique designed to efficiently index Boolean expressions over a high-dimensional continuous space. At the core of BE*-Tree lie two innovative ideas: (1) a bi-directional tree expansion build as a top-down (data and space clustering) and a bottom-up growths (space clustering), which together enable indexing only non-empty continuous sub-spaces, and (2) an overlap-free splitting strategy. Finally, the performance of BE*-Tree is proven through a comprehensive experimental comparison against state-of-the-art index structures for matching Boolean expressions.

1.3.4 Data Indexing

Multiversion databases store both current and historical data. Rows are typically annotated with timestamps representing the period when the row is/was valid. We develop novel techniques for reducing index maintenance in multiversion databases, so that indexes can be used effectively for analytical queries over current data without being a heavy burden on transaction throughput. To achieve this end, in Chapter 6, we design persistent data structures in the storage hierarchy to employ an extra level of indirection. The indirection level is stored on solid state drives (SSDs) that can support very fast random I/Os, so that traversing the extra level of indirection incurs a relatively small overhead.

The extra level of indirection dramatically reduces the number of magnetic disk I/Os that are needed for index updates, and localizes maintenance to indexes on updated attributes. Further, we batch insertions within the indirection layer in order to reduce physical disk I/Os for indexing new records. By reducing the index maintenance overhead on transactions, we enable operational data stores to create more indexes to support queries. We have developed a prototype of our indirection proposal by extending the widely used Generalized Search Tree (GiST) open-source project, which is also employed in Postgres. Our working implementation demonstrates that we can significantly reduce index maintenance and/or query processing cost, by a factor of 3. For insertions of new records, our novel batching technique can save up to 90% of the insertion time.

1.3.5 XML/XPath-to-Kernel Mapping

Content-based architectures for XML data dissemination are gaining increasing attention both in academia and industry. These dissemination networks are the building blocks of selective information dissemination applications, which have wide applicability such as sharing and integrating information in both scientific and corporate domains. At the heart of these dissemination services, there is a need for a fast engine to match an incoming XML message against stored XPath expressions to determine interested message consumers. In Chapter 7, we formalize the mapping of XML/XPath expressions into predicated-based Boolean expressions and present a dichotomy for the expressiveness of our mapping; parts of this chapter are published in [175, 174]. Our proposed mapping enables the use of existing state-of-the-art Boolean expression matchers (i.e., query indexing kernel).

1.3.6 Data-centric Workflows-to-Kernel Mapping

In Chapter 8, we develop an approach for safe distribution and parallel execution of data-centric workflows over the publish/subscribe abstraction; parts of this chapter are published in [174]. In essence, we
present a novel re-formulation of data-centric workflows that is designed to utilize the loosely coupled and distributed nature of publish/subscribe systems. Furthermore, we argue for the practicality and expressiveness of our approach by mapping an industry-based data-centric workflow, namely, IBM Business Artifacts with Guard-Stage-Milestone (GSM), into the publish/subscribe abstraction. In short, the contributions of this chapter are three-fold: (1) mapping of data-centric workflow into publish/subscribe to achieve distributed and parallel execution; (2) detailed theoretical analysis of our mapping; and (3) formalizing the complexity of optimal workflow distribution over the publish/subscribe abstraction.

### 1.3.7 FPGA Query Indexing

In Chapter 9, we present *fpga-ToPSS* (Toronto Publish/Subscribe System), an efficient event processing platform to support high-frequency and low-latency event matching. *fpga-ToPSS* is built over reconfigurable hardware – FPGAs – to achieve line-rate processing by exploring various degrees of parallelism; parts of this chapter are published in [180, 182, 183, 174]. Furthermore, each of our proposed FPGA-based designs is geared towards a unique application requirement, such as flexibility, adaptability, scalability, or pure performance, such that each solution is specifically optimized to attain a high level of parallelism. Therefore, each solution is casted as a design trade-off between the degree of parallelism versus the desired application requirement. Moreover, our query indexing kernel on FPGAs supports Boolean expression matching with an expressive predicate language applicable to a wide range of applications including real-time data analysis, algorithmic trading, targeted advertising, and (complex) event processing.

### 1.3.8 FPGA SQL Query Indexing

In Chapter 10, we present an efficient SQL query indexing kernel to support query processing over high-frequency event streams; parts of this chapter are published in [181, 174]. Our kernel is accelerated using reconfigurable hardware – FPGAs – to achieve line-rate multi-query processing by exploiting unprecedented degrees of parallelism and potential for pipelining, only available through custom-built, application-specific and low-level logic design. In particular, we reveal key opportunities for extracting potential intra- and inter-parallelism to enhance multi-query optimization on hardware. Moreover, such a multi-query event stream processing engine is at the core of a wide range of applications including real-time data analytics, algorithmic trading, targeted advertising, and (complex) event processing.

### 1.4 Thesis Organization

The rest of this thesis is organized as follows. We begin by discussing the related work and position this thesis with respect to state-of-the-art (cf. Chapter 2). We present the core of our query indexing kernel, namely, *BE-Tree* (an in-memory resident structure), and provide detailed theoretical and experimental analysis of *BE-Tree* (cf. Chapter 3). After establishing the dominance of *BE-Tree*, we shift our focus to parallelizing *BE-Tree* (using multi-core architecture) by introducing a novel adaptive compressed matching algorithm (*A-PCM*) and discussing our comprehensive evaluation (cf. Chapter 4). Subsequently, we enhance our indexing kernel to support an effective *top-k* processing by developing *BE*-Tree (cf. Chapter 5). We conclude our indexing kernel by proposing a novel *Indirection* indexing technique that leverages...
SSDs to improve index maintenance for multi-version databases followed by an extensive theoretical and empirical analysis (cf. Chapter 6).

The second part of this thesis is dedicated to extending our query indexing kernel expressiveness. We propose an XML/XPath mapping into our query indexing kernel, in which XPath queries are expressed as Boolean expressions and XML documents as attribute-value pairs (cf. Chapter 7). Next, we extend our kernel formalism, based on the publish/subscribe abstraction, to theoretically support parallel and distributed execution of data-centric workflows motivated by a real industry workflow formalism (cf. Chapter 8).

In the third part of this thesis, we turn to hardware acceleration to achieve high-throughput and low-latency query indexing kernel. Using FPGAs, we propose a novel low-level, custom-design logical circuits to accelerate Boolean expressing indexing (cf. Chapter 9). Finally, we identify opportunities for accelerating SQL query indexing on FPGAs (cf. Chapter 10).

Lastly, the conclusions and future research directions of this thesis are discussed (cf. Chapter 11).
Chapter 2

Related Work

This chapter, we survey the existing approaches related to this thesis, and we discuss the importance of our contributions, and how these contributions are positioned with respect to prior work.

2.1 Query Indexing

Problems related to indexing Boolean expressions have been studied in many contexts: Expert systems [90, 83], active databases [101], trigger processing [100, 212], publish/subscribe matching [220, 12, 74, 38, 140, 36, 211, 82, 77], and XPath/XML matching (e.g., [67, 45, 39, 124, 125, 148, 175]). Indexing in multi-dimensional space has been extensively studied (e.g., [27, 99, 26, 189, 28, 59, 87]).

The work on expert systems, active databases, and trigger processing [90, 83, 212, 101, 100] as well as certain publish/subscribe (pub/sub) work [36] focus on language expressiveness and not on scaling to thousands of dimensions and millions of expressions. XPath/XML matching [67, 45, 39, 124, 125, 148] is based on a completely different language from what BE-Tree. These approaches are therefore not directly applicable, and in this section, we concentrate our review on pub/sub matching [220, 12, 74, 38, 172, 184, 81, 140, 211, 77, 176, 142, 58, 143, 179].

Publish/Subscribe Matching

Two main categories of matching algorithms have been proposed: counting-based [220, 74, 211, 77, 142, 58, 143] and tree-based [12, 38] approaches. Furthermore, existing work can be further classified as, either key-based in which for each expression a set of predicates are chosen as identifier [74], or as non-key based [220, 38, 211, 77].\(^1\) Counting-based methods aim to minimize the number of predicate evaluations by constructing an inverted index over all unique predicates. The two most efficient counting-based algorithms are Propagation [74], a key-based method, and the \(k\)-index [211], a non-key-based method. Similarly, tree-based methods are designed to reduce predicate evaluations and to recursively divide search space by eliminating subscriptions on encountering unsatisfiable predicates. Tree-based methods are proven to outperform counting-based algorithms [120]. Despite this theoretical result, only few efficient tree-based matching algorithms exist. The most prominent tree-based approach, Gryphon, is a

\(^1\)The Access Predicate Pruning (APP) [77] introduces a filtering strategy using the notion of access predicate, but APP is considered a non-key based method under our strict classification.
static, non-key based method [12]. Our proposed BE-Tree is a novel tree-based approach, which also employs keys, that we show to outperform existing approaches [220, 12, 74, 211, 77].

The Propagation algorithm is the state-of-the-art counting-based method with two main strengths [74]. First, a typical counting-based inverted index is replaced by a set of multi-attribute hashing schemes; each multi-attribute hashing scheme is referred to as an access predicate (i.e., key). Second, keys are selected from a candidate pool using an effective cost-based optimization tuned by the workload distribution [74]. The weaknesses of Propagation are as follows: limiting keys to only a small set of equality predicates in order to use hashing and to avoid exponential blow up in the number of candidate keys; assuming that subscriptions are uniformly distributed across keys to avoid degeneration of hashing into a sequential scan over subscriptions; and maintaining a large collection of candidate hash configurations, using histograms, based on a greedy selection. Our proposed BE-Tree structure overcomes all these shortcomings by employing a multi-layer structure to avoid hashing degeneration and to enable on-demand creation of histograms as needed; a self-adjusting mechanism to adapt to workload changes without maintaining histograms; and, lastly, to support a rich set of operators beyond the equality predicate.

To enrich Propagation with interval predicate, a Hierarchical Clustering (HC), using a variant of the Propagation cost function, is proposed in [184]. However, the problem of candidate generation is worsened in HC because each cluster must now maintains a complete set of histograms for all dimensions, e.g., dimension $d = 10^3$, cluster size of $10^2$, and $5 \times 10^6$ subscriptions, roughly $5 \times 10^7$ histograms is needed [184]. Furthermore, HC dynamics, merging and splitting, are only local operations between a leaf and its parent and HC global structure fails to adapt to workload changes [184]. HC has also shifted its focus to a more general disk-based indexing (as opposed to main memory indexing) that scales to only tens of dimensions, but, most important, HC disregards the key observation that subscriptions tend be defined over a discrete and a finite domain [220, 12, 74, 43, 211]. This key domain property introduces new challenges, yet it provides a unique opportunity for further exploitation of the inherit structure, which is fully leveraged in BE-Tree.

The latest advancement in the counting-based algorithm is $k$-index [211], which gracefully scales to thousands of dimensions and supports equality predicates ($\in$) and non-equality predicates ($\notin$). $k$-index partitions subscriptions based on their number of predicates to efficiently prune subscriptions with too few matching predicates; however, $k$-index is static and does not support dynamic insertion and deletion. What distinguishes BE-Tree from $k$-index is that BE-Tree is fully dynamic, naturally supports richer predicate operators (e.g. range operators), and adapts to workload changes.

Other, complementary techniques to enhance pub/sub matching are event batch processing [81] and top-$k$ matching [140, 211]. The former reduces the number of index lookups by batching similar events. The latter aims to improve matching by only returning the top-$k$ matching subscriptions.

Another emerging area of research is to improve language expressiveness. For example, the subscription languages in [38, 82] support both Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF), while the $k$-index supports either CNF or DNF subscription language [211]. Finally, there is a paradigm shift (symmetric pub/sub) in which events producers are also able to impose filtering conditions on events’ subscribers [172], which substantially improves the expressive power of the event language; this new paradigm is supported by our BE-Tree and proposed matching semantics.

Orthogonal to matching problem is the distributed content-based routing and subscription propagation algorithms (e.g., [204, 116]). Many of these approaches employ novel techniques based on Bloom
filters. However, our usage of Bloom filter is complementary to the core of BE-Tree algorithm and serves only as additional optimization layer for filtering the subscriptions stored in the leaf levels. Similar pruning technique is also explored in [142, 58, 143].

**Traditional Multi-dimensional Indexing**

An alternative approach in building pub/sub matching engine is to use the multidimensional indexing developed in the database community; the most prominent multidimensional structure is $R$-tree [99], which supports indexing spatial extended objects, theoretically a suitable index to solve the pub/sub matching problem. The $R$-tree introduced the idea of overlapping partitions to achieve high space utilization properties, a desirable disk-based property, at the cost of downgrading the retrieval performance [99]. However, this overlapping side effects further worsens as the dimensionality increases (above three) at which point a sequential scan is more efficient [28]. The problem of reducing overlapping partitions has been tackled from different angles: $R^+$-tree reduces overlapping by clipping objects, but it results in exponential space blow up [189]; $R^*$-tree delays splitting and relies on re-insertion and attempts to geometrically minimizes the overlap during splitting and insertion [26]; X-tree is a hybrid of sequential scan and $R^*$-tree and switches to sequential scan when no overlap-free split exist [28]. The X-tree is the only structure that scales well to tens of dimensions, yet takes only a passive approach to solve the overlapping problem by exploiting only the physical storage property, i.e., favoring sequential v.s. random access.

Among many others, interval indexing approaches such as Segment Tree [29], Interval Tree [29], and $R$-tree [99] have been proposed to index one-dimensional objects. The Segment Tree and Interval Tree are static structures. Although $R$-tree is a dynamic structure, it is sensitive to the insertion sequence, while BE-Tree was designed to be independent of insertion sequence.

Matching in high-dimensional space, diverges from classical database indexing in four important ways. (1) BE-Tree has to cope with data of much higher dimensionality (order of thousands), that is orders of magnitude larger than capabilities of existing high-dimensional indexing structures [99, 28, 87]. (2) Expressions indexed by BE-Tree impose restrictions on a small subspace only and are fully defined everywhere else. As a result, there is high degree of overlap among expressions which renders current indexing techniques inapplicable. For instance, X-Tree [28], often the most suitable index for high-dimensional data, degenerates to a sequential scan in situations where all expressions overlap. (3) Much high-dimensional indexing work focuses on high space utilization and reducing random accesses, as opposed to optimize matching time (lookup); disk is assumed as storage medium and disk I/O is the bottleneck. BE-Tree, on the other hand, is a main memory structure. (4) BE-Tree aims to support discrete, finite domains, while many high-dimensional indexing structures are designed for continuous unbounded domains that are unable to benefit from the finite and discrete domain structure.

### 2.2 Parallel Query Indexing

The problem related to indexing Boolean expressions has been studied in the database (e.g., [99, 26, 87]) and the publish/subscribe (e.g., [220, 12, 74, 67, 38, 218, 36, 10, 211, 82, 176, 76, 178]) communities are different in two important ways. First, the database indexing solves the reverse problem: in the database context, querying means finding the relevant tuples (events) for a given query (subscription), but in event processing context, matching (through indexing) means finding the relevant subscriptions
Chapter 2. Related Work

(queries) for a given event (tuple). Second, publish/subscribe matching algorithms overlook both parallel event matching and event stream re-ordering, which is central in exploiting the exponential growth trend in the number of cores of modern hardware [117, 166].

One of the latest publish/subscribe indexing structure is our BE-Tree, a key-based approach, that introduces a two-phase space-cutting abstraction that is proven to outperform all existing techniques [176, 179]. Despite BE-Tree’s effectiveness, it provides no guarantee for exactly-once predicate evaluation. More importantly, there are no attempts to parallelize BE-Tree, which is a non-regular, tree-based structure that suffers from the well-known pointer-chasing and tree traversal issues that inevitably incur cache-misses. This problem is further amplified as the number of parallel tree traversal are increased, resulting in a higher number of cache-misses. In addition, BE-Tree fails to algorithmically exploit the large number of cores and large shared caches prevailing in multi-core architectures [166]. To address these shortcomings, which are not limited to only BE-Tree, we present novel adaptive parallel compressed event matching (A-PCM) and online stream re-ordering (OSR) algorithms that exploit all these new hardware properties.

Finally, online sorting and stream re-ordering in database indexing and in the storage context has relied heavily on locality-sensitive hashing [113] in order to cope with the curse of dimensionality (e.g., [91, 139, 85]). In general, the main shortcoming of locality-sensitive hashing, which limits its applicability in practical settings, is the complexity and uncertainty of tuning locality-sensitive hashing parameters [69]. These parameters are highly data dependent, which complicates the performance tuning approaches [69]. In contrast, in this thesis, we present our OSR algorithm for high-dimensional data that not only reduces the parameter tuning challenge, but, more importantly, solves the two major challenges unaccounted by locality-sensitive hashing, namely, handling incomplete event data and efficient reasoning about stream heterogeneity.

2.3 Top-k Query Indexing

As mentioned previously, the database literature focuses on much lower dimensionality, while we target a dimensionality in order of thousands, which is orders of magnitude larger than capabilities of existing database indexing structures [87]. More importantly, the database top-k processing [75, 112] differs from our proposed top-k model in an important respect: our top-k model solves the reverse problem. In the database context, top-k querying means finding the most relevant tuples (events) for a given query (subscription). But in our context, top-k matching means finding the most relevant subscriptions (queries) for a given event (tuple).

In addition to the two categories of matching algorithms, namely, counting-based [220, 74, 211] and tree-based [12, 38, 176], these algorithms could further be classified either as rigid- [220, 74, 211, 176, 179] or non-rigid-based, our proposed BE*-Tree, [178]. What constitutes rigidity is a pre-determined clustering of the space. For example, for a skewed distribution in which all data is clustered in a small subspace, rigid clustering continues splitting the space in half, irrespective of where the data is actually located, and is unable to avoid indexing the empty subspace.

Our proposed BE*-Tree [178], unlike its predecessor BE-Tree [176, 179], operates on a non-rigid clustering through a self-adjusting mechanism using the bi-directional tree expansion and the overlap-free splitting strategy that together let BE*-Tree adapt to skewed workloads, cluster only non-empty subspaces, and improve search space pruning. Above all, BE*-Tree enables the top-k relevance-match-determination model. Although BE*-Tree addresses the rigidity and top-k limitations of BE-Tree, yet BE-Tree has two
notable advantages over BE*-Tree: it is algorithmically easier to implement and robust w.r.t. insertion sequence (a desirable property also lacking in R-Tree family of indexes [87]).

A fresh look at enhancing existing pub/sub matching algorithms is to leverage top-k processing techniques, which not only concentrate on determining the most relevant matches, but can also improve matching performance. An early top-k model is presented in [140]; however, this model is based on a fixed and predetermined scoring function, i.e., the score for each expression is computed independent of the incoming event. In addition, this approach is an extension of the R-Tree, the interval tree, or the segment tree structure; as a result, it has difficulties scaling beyond a few dimensions [140]. In contrast, a scalable top-k model, but based on a static and flat structure, with a generic scoring function, which also takes the event into consideration, is introduced in k-index [211]. BE*-Tree’s top-k model not only scales to thousands of dimensions but introduces a hierarchical top-k model. Our top-k model supports a generic scoring function, that achieves an effective pruning power for determining the most relevant matches.

BE*-Tree’s hierarchical top-k structure differs from the flat structure of k-index [211]. This key structural differentiation is depicted in Figure 2.1. A hierarchical model achieves enhanced pruning because it refines upper bound scores as more expressions are seen (as traversing down the tree). Such pruning is not possible with a flat structure, in which the upper bound score is computed once at the top-layer because as more expressions are seen, the new information is not reflected back in the original top-layer scores.

2.4 Data Indexing and SSDs

There has been extensive work on storing high value data on SSDs. Some of these studies target the problem of data placement in relational databases to take better advantage of the SSD characteristics. In [40] database objects are either placed on HDDs or SSDs based on workload characteristics. As opposed to using SSDs and HDDs at the same level in the storage hierarchy, SSDs are also used as a second layer cache between main memory and HDDs [121, 31, 41, 68].

The use of an SSD cache with a write-back policy would not solve our problem as effectively as the Indirection technique. One would need space to hold entire indexes if one wants to avoid HDD index I/O for random updates and insertions. In contrast, our method avoids HDD I/O with a much smaller SSD footprint. The Indirection technique and SSD caching of HDD pages are complementary, and can
be used in tandem. In fact, using indirection improves the cache behaviour by avoiding reading/writing unnecessary pages (i.e., it avoids polluting the cache).

Adding a level of indirection is a commonly used programming technique relevant to a variety of systems problems [16]. The kind of indirection we propose in this paper is used in log-structured file systems [173, 105] but only at page granularity. We use indirection at record granularity. The log-structured storage is also exploited in database management systems [210, 159, 188], but no indirection layer is used; thus, the common merging and compaction operations in log-structured storage result in expensive rebuilding of all indexes. By employing our indirection technique this index rebuilding can be avoided.

Page-level indirection tables are also used to improve the lifespan of SSDs. In [52], a system called CAFTL is proposed to eliminate duplicate writes on SSDs. By keeping a mapping table of blocks the redundant block writes on SSDs are eliminated.

In [217], Wu et al. proposes a software layer called BFTL to store B-tree indexes on flash devices efficiently. IUD operations cause significant byte-wise operations for B-tree reorganization. The proposed layer reduces the performance overhead of these updates on the flash device by aggregating the updates on a particular page.

In [70], Dou et al. propose specialized index structures and algorithms that support querying of historical data in flash-equipped sensor devices. Since the sensor devices have limited memory capacity (SRAM) and the underlying flash devices have certain limitations, there are challenges in maintaining and querying indexes.

The deferral of index I/Os is used in several recent papers on improving index performance [30, 158, 9]. In those papers, changes are accumulated at intermediate nodes, and propagated to children in batches. Searches need to examine buffers for keys that match. This line of work is complementary but similar to our LIDBlock method in that both techniques buffer insertions to amortize physical I/O.

SSDs are used to support online updates in data warehouses [19]. Incoming updates are first cached in SSDs and later merged with the older records on HDDs to answer queries. In [19], data records are accessed primarily through table scans rather than indexes.

Many specialized indexes for versioned and temporal data have been proposed. A comprehensive survey of temporal indexing methods is provided in [185]. Tree based indexes on temporal data include the multiversion B-tree [25], Interval B-tree [15], Interval B+-tree [35], TP-Index [190], Append-only Tree [96] and Monotonic B+tree [72]. Efficiently indexing data with branched evolution is discussed by Jouni et al. [119], who build efficient structures to run queries on both current and historical data.

Specialized transaction time database systems such as Immortal DB [136, 137] provide high performance for temporal applications. Lomet et al. [137] describe how a temporal indexing technique, the TSB-tree, is integrated into SQL Server. The paper also describes an efficient page layout for multiversion databases.

### 2.5 XML/XPath Processing

The related work can be broadly classified into publish/subscribe matching algorithms and XML/XPath matching (i.e., XML filtering) algorithms.

Much work has been devoted to the development of matching algorithms in the context of publish/subscribe systems. Our work builds on these algorithms [220, 94, 12, 74, 18, 130, 211, 82, 176, 178]
in order to utilize these techniques for XPath/XML matching. However, none of these algorithms ad-
dresses techniques to process XML message against XPath expressions or even hints at tree-structured
data processing. All the above approaches assume sets of attribute-value pairs as events and conjunctive
Boolean predicate formulas as subscriptions.

A large body of work have been dedicated to the XML/XPath matching problem [167, 67, 45, 98,
95, 37, 39, 106, 131, 124, 125, 148]. However, the majority of these approaches do not pursue casting the
XML/XPath matching problem as the pub/sub matching problem, with the exception of [167, 106, 175].
On the contrary, the majority of these approaches introduce new techniques that can be broadly classified
into automaton-based algorithms and index-based ones.

The automaton-based approaches [67, 95, 98, 39], most prominently YFilter [67], build automata
based on the XPath expressions in the system, while [39] also employs an NFA pruning strategy based
on prefix and suffix overlap of XPath expressions. Other automaton-based approaches mostly focus on
twig patterns. pFiST introduces value-based filtering [124], iFist proposes a holistic matching of twig
patterns using a bottom-up approach [125], and BoXFilter also uses a bottom-up approach, but uses the
Prüfer sequence for sequencing twig patterns [148].

The index-based approaches [167, 37, 45, 106] take advantage of precomputed schemes on either the
XML documents or the XPath expressions. The index-based approaches are further extended in [106]
toward the idea of predicate-based encodings. However, this approach heavily relies on special-purpose
data structures employing a complicated ordering semantics to maintain predicate inter-relationships
during the matching process [106]. Also, the encoding chosen differs from the one developed in this
chapter. In [175], we showed that GPX-Matcher outperforms both BPA (called basic-pc-ap in [106]) and
the automaton-based Yfilter [67].

YFilter is a well-known algorithm for matching XML documents against XPath expressions [67]. It
builds a non-deterministic finite automaton (NFA) from all the XPath expressions in the system. The
final states of the NFA are associated with list of expressions. The parsing of the XML message, one
node at a time, triggers the transitions in the NFA. Whenever a final state is reached, it means that the
corresponding expressions are matched by the incoming XML message. The only difference between a
traditional NFA and the YFilter data structure is that the execution of the NFA does not terminate as
soon as the first final state is reached, but it continues until all possible accepting states are reached;
thereby discovering all matching queries.

Other XML dissemination research [194, 79, 46, 122] focuses on outlining the idea of XML-based
content-dissemination [79], addresses the problem of failure handling in these systems [46], addresses
the problem of retrieving XML documents based on input XPath expressions [122], and discusses the
design and implementation of an XML router [194], which is closely related to the XPath/XML matching
problem. However, none of these approaches proposes a matching algorithm for efficiently evaluating
incoming XML messages against the stored XPath expressions.

Content-based routing protocols have been developed in [22, 161, 42, 57, 80, 114]. However, none
of these protocols addresses the problem of determining the routing decision for tree-structured data,
such as XML. Therefore, our work is complementary to these approaches and enables the use of these
protocols for the content-based routing of XML data.
2.6 FPGAs Acceleration

An FPGA is a semiconductor device with programmable lookup-tables (LUTs) that are used to implement truth tables for logic circuits with a small number of inputs (on the order of 4 to 6 typically). FPGAs may also contain memory in the form of flip-flops and block RAMs (BRAMs), which are small memories (a few kilobits), that together provide small storage capacity but a large bandwidth for circuits in the FPGA. Thousands of these building blocks are connected with a programmable interconnect to implement larger-scale circuits.

Past work has shown that FPGAs (and GPUs) are a viable solution for building custom accelerated components [138, 89, 147, 150, 200, 149, 151, 215, 152, 182, 199, 142, 214, 216, 192, 33, 201, 58, 143]. For instance, [147] demonstrates a design for accelerated XML processing, and [149] shows an FPGA solution for processing market feed data. As opposed to these approaches, our work concentrates on supporting an FPGA-based middleware for general event processing applications specifically designed to accelerate the event matching computation. Similarly, [151, 199] presents a database query processing framework that uses FPGAs to efficiently run hardware-encoded queries while [215] demonstrates the running of hardware-encoded regular expression queries; the key insight was the realization that the deterministic finite automata (DFA), although suitable for software solutions, results in an explosion of space; thereby, to bound the required space on the FPGA, a non-deterministic finite automata (NFA) is utilized in [192, 33]. Our approach differs from [151, 215, 199] as we primarily focus on developing FPGA-based middleware (as opposed to running a single or multi database query on hardware) for large scale event processing applications that scale to thousands of subscriptions on a single FPGA chip, while stacking multiple FPGA chips enables the scaling to millions of subscriptions.

For instance, our work in [180, 182, 183], described in Chapter 9, focuses on atomic and stateless matching (i.e., select queries). However, our work in [181], presented in Chapter 10, concentrates on supporting a more general multi-query stream processing (stateful matching) specifically designed to accelerate the execution of SPJ queries. Alternatively, [151] presented an efficient FPGA implementation of a single query (without join) while [199] focused on the data flow for streaming join over a large window size that spans many processing cores. Our approach also differs from [151, 199] as we are primarily concerned with multi-query optimization (with joins computed over a moderate size window) using Rete-like processing networks, supporting a rich relational algebra over event data streams, and offering an unprecedented degree of inter- and intra-operator parallelism that is only available through low-level logic design. Furthermore, the join processing technique proposed in [199], models join as an efficient data flow problem over a large number of processing cores, a higher-level view of the problem, while our join processing technique in [181] focuses on low-level implementation of join computation within each processing core. Thus, these two techniques can be seen as complementary.

Other related research projects in the area of high-speed pattern matching for network intrusion detection, high-speed router architecture design, and longest-prefix matching [222] are based on Content Addressable Memory (CAM) and Ternary CAM (TCAM). These hardware memories are software equivalent associative arrays that provide the ability to search the entire content of the memory in one clock-cycle using either an exact pattern string (in CAMs) or a partial pattern with wild-card search (or do not care states) using a TCAM. However, both the above approaches using CAMs and TCAMS currently offer only a limited range of scalability for large data sets due to notoriously high power consumption and hardware resource costs associated with their operation and implementation [141, 164]. Similarly, other research projects [195, 66] have utilized bloom filters, that are also associative arrays but only store
signature of the actual content that is generated using hash functions. Use of Graphic processor Units (GPUs) to accelerate pattern matching algorithms is also an active area of research [142, 58, 58, 143, 205]. However, utilizing GPUs poses the particular challenge of transforming a given algorithm to utilize the highly parallel architecture that is only designed to perform massive matrix data manipulation at high throughputs. Finally, the advancement in hardware memory devices based on photonics [24] and spintronics [168] technologies will undoubtedly pave the way for the next generation of high-speed pattern matching and database query processing.

On a different front, a recent body of work has emerged that investigates the use of new hardware architectures for data management systems [102]; for instance, multi-core architectures were utilized to improve the performance of a database storage manager [118] and to enhance transaction and query processing [165].

2.7 Data-centric Workflow Execution

Our work is based on a data-centric business artifacts paradigm [157, 32, 50] with GSM meta-model being a natural evolution from the earlier practical artifact meta-models [54, 198], but using a declarative basis, and supporting modularity and parallelism within artifact instances. The existing work on GSM operational semantics does not consider distributed execution [64]. Recently different data-centric approaches have been proposed including a FlexConnect metamodel [171], in which processes are organized as interacting business objects; the Case Management paradigm [207, 65]; and the AXML Artifact model [7, 8], which is based on a declarative form of artifacts using Active XML as a basis [6].

There exists a body of work focused on various aspects of distributed workflow execution. For instance, [23] has a similar goal as our work but is applied to an inherently activity-centric workflow models, in which data is only considered as input and output (dataflow) of flow activities and no data-centric execution is supported. This is also true in [21], in which scheduling of workflows in self-organizing wireless networks is addressed to respect resource allocation constraints and dynamic topology changes, or for [187, 133, 155] that use publish/subscribe techniques to implement some of the BPM execution aspects.

Distributed workflow processing has been studied in the 1990s to also address scalability, fault resilience, and enterprise-wide workflow management [13, 213, 154]. A detailed design of a distributed workflow management system was proposed in [13]. The work bares similarity with our approach in that a business process is fully distributed among a set of nodes. However, the distribution architectures differ fundamentally. In our approach, a content-based message routing substrate is built to naturally enable task decoupling, dynamic reconfiguration, system monitoring, and run-time control. This is not addressed in the earlier work.

A behavior preserving transformation of a centralized activity chart, representing a workflow, into an equivalent partitioned one is described in [154] and realized in the MENTOR system [213]. MENTOR is inspired by compiler-based techniques, including control flow and data flow analysis, in order to parallelize the business process [156]. However, these approaches are complementary to our work since we operate with the original business process model without analyzing the process. An advantage of executing an unmodified process is that dynamic changes to the executing business process instances are possible, as their structure remains unchanged from the original specification.

Finally, an approach to integrate existing business processes as part of a larger workflow is presented
in [44]. The authors define event points in business processes where events can be received or sent. Events are filtered, correlated, and dispatched using a centralized publish/subscribe model. The interaction of existing business processes is synchronized by event communication. This is similar to our work in terms of allowing business processes to publish and subscribe. In our approach, activities in a business process are decoupled, and the communication between them is performed in a content-based publish/subscribe broker network.
Part II

Kernel Core
Chapter 3

Query Indexing (Main Memory)

3.1 Introduction

The efficient indexing of Boolean expressions is a common problem at the center of a number of data management applications. For example, for event processing and publish/subscribe, Boolean expressions represent events and subscriber interests [12, 74, 38, 153, 211], for online advertising and information filtering, Boolean expressions represent advertiser profiles and filters [140, 211, 82], and for approximate string matching they can represent string patterns [78, 51, 47]. In all scenarios, key requirements are the scaling to millions of expressions and to sub-second matching latency. We use a data management scenario for co-spaces as in-depth example. Co-spaces are an emerging concept to model the co-existence of physical and virtual worlds touted by the Claremont Report as an area of rising interest for database researchers [11, 160]. Consider, for example, a mobile shopping application, where a shopper enters a physical mall and her mobile device submits her shopping preferences (i.e., subscriptions) to the virtual mall database, as follows: 

\[
\text{genre} = \text{classics, } \text{era} \in \{1920s, 1950s\}, \text{price BETWEEN } [20, 40], \text{ranking} < 5, \text{format} \notin \{\text{mass market, paperback}\}.
\]

Now, assume a new promotional item (i.e., an event) matches the shopper’s interests and the item detail is pushed to her mobile device. An example of a matching item is as follows: 

\[
\text{genre} = \text{classics, title} = 'Fahrenheit 451', \text{author} = \text{Ray Bradbury, era} = 1950s, \text{price} = 26, \text{ranking} = 2, \text{format} = \text{hardcover}.
\]

In the example, both the shopper’s interest and the promotional item are defined over multiple attributes (i.e., dimensions in space) such as genre and price in which each attribute has a discrete and finite domain. Furthermore, each attribute of interest is constrained to a set of values with an operator. The triple consisting of attribute, operator, and set of values is referred to as a Boolean predicate. A conjunction of Boolean predicates, which here represents both the shopper’s interest and the promotional item, is a Boolean expression. Next, we present the broad applicability of Boolean expression indexing matching; followed by the shortcoming of the existing techniques.

Motivation

**Online Profile Matching** Prominent profile-driven Web applications are targeted Web advertising (e.g., Google, Microsoft, Yahoo!) and job seeker sites (e.g., Monster). For example, in Web advertising, a demographic targeting service specifies constraints such as \(\text{age} \in \{25, 27, 29\}\) while an incoming user’s profile also includes information such as \(\text{age} = 27\). Thus, only ads that match a user profile are displayed. Similarly, in online job sites, an employer submits the job detail, \(\text{category} = \text{‘green jobs’}, \text{hours/week} = \text{30}\)
Chapter 3. Query Indexing (Main Memory)

> 15, and \(rate < 45\), while a job seeker registers his profile, \(category = \text{‘green jobs'}, hours/week = 20, \text{and} rate = 30\), in which the employer is notified of only matching applicants \([140, 211, 82]\). These scenarios require the indexing of potentially millions of expressions and require event matching latency in sub-second.

**Complex Event Processing** Event processing is gaining rising interest in industry and in academia. The common application pattern is that event processing agents publishes events while other agents subscribe to events of interest. Extensive research has been devoted to developing efficient and scalable algorithms to match events and subscriber’s interests \([220, 12, 74, 38, 172, 218, 211, 82, 77, 142]\). The predominant abstraction used in this context, is the content-based publish/subscribe paradigm to model an event processing application. Applications that have been referenced in this space include algorithmic trading and (financial) data dissemination \([180, 183]\), business process management \([107, 133]\), sense-and-respond \([48]\), intrusion detection system \([56, 77]\), and location-based services \([219]\).

**Data Quality** Data quality has been an active area of research in the database community over the past decade \([78, 51, 47]\). In general, data quality is effected by typing mistakes, lack of standards and integrity constraints, and inconsistent data mappings resulting in different representations of identical entities. Therefore, many approximate string matching algorithms have been proposed to identify similar entities \([51, 47]\). These algorithms are based on tokenization of a string into a set of \(q\)-grams (a sequence of \(q\) consecutive characters). For example, a 3-gram tokenization of “string” is given by \{'str’, ‘tri’, ‘rin’, ‘ing’\}. One of the main challenges for a \(q\)-gram transformation is the curse of dimensionality \([51, 47]\). That is 3-grams result in at least \(26^3\) dimensions. For a realistic BE-Tree evaluation over real-world data, we propose a representation of a set of \(g\)-grams as Boolean expressions to significantly reduce dimensionality and leverage BE-Tree to actually solve the approximate substring matching problem.

Next, we discuss how \(g\)-grams are converted to Boolean expressions, which similar to above scenario demands scalability to large expression sets is paramount.

String tokenization using \(q\)-grams maps the string into a high-dimensional vector space model, in which the domain of each dimension is binary. The size of this space is exponential in the length of the \(q\)-gram. For instance, a \(q\)-gram of size three results in a space with \(26^3\) dimensions.

The vector space model representation of a tokenized string (e.g., \{'str’, ‘tri’, ‘rin’, ‘ing’\}) can be expressed by setting dimensions associated to each of its grams (e.g., ‘str’, ‘tri’, ‘rin’, and ‘ing’) to 1 and everything else to 0. Alternatively, we can concisely express this as binary equality predicates, e.g., the predicate with dimension (i.e., attribute) ‘str’ must be equal to 1 and ignoring dimensions with value 0. Therefore, instead of expressing a tokenized string as a vector of 0s or 1s, we can express it as a set of equality predicates. We can also take this one step further by going beyond binary domains to capture more interesting relationships among \(q\)-grams, and as a byproduct, reduce the space dimensionality.

The original model mapped each \(q\)-gram to a dimension, instead we can map only the prefix of each \(q\)-gram to a dimension, and map the rest as a value to the corresponding dimension. For example, the \(q\)-gram ‘str’ is mapped to the equality predicate \(\text{‘st’ = ‘r’}\), ‘st’ now represents the predicate’s dimension and ‘r’ is mapped to the value in this dimension.

In this new mapping, the number of dimensions is reduced from \(26^3\) to \(26^2\), and a new opportunity to express similarity among overlapping \(q\)-grams results. For example, the \(q\)-grams ‘str’ and ‘ste’ are now both mapped to dimensions represented by ‘st’ (already signifying a similarity among these two \(q\)-grams), but also the value ‘r’ and ‘e’ can play an important role. For instance, since the letters ‘r’ and ‘e’ are adjacent on a standard U.S. keyboard, it is possible, that due to a typing error, the \(q\)-gram ‘str’
Applications in the Co-spaces

The co-existence of virtual and physical worlds brings unique opportunities for a new generation of applications. Applications that use information gathered from the virtual world to continuously enrich a users’ physical world experience while using the real-time information gathered from the physical world to refresh the virtual world in turn \[11, 160\]. Examples of co-space applications are marketplace applications that allow virtual and physical shoppers to compete (bid on the last item) or cooperate (buy one get one free), location-based gaming that changes the gamer’s environment relative to the gamer’s physical location, and social networking applications that detect when virtual friends are within a close proximity and initiates a physical interaction among them \[160\]. We already illustrated a detailed example of where and how expression indexing is important in this context in the introduction of this chapter.

Boolean Expression Matching Challenges

There are four major challenges to efficiently index Boolean expressions. First, the index structure must scale to millions of Boolean expressions defined over a high-dimensional space and afford efficient lookup (i.e., expression matching). Second, the index must support predicates with an expressive set of operators. Third, the index must enable dynamic insertion and deletion of expressions. Fourth, the index must adapt to changing workload patterns.

However, existing techniques are inadequate to satisfy these four requirements. For instance, techniques used in expert and rule-based systems support expressive predicate languages \[83\], but are unable to scale to millions of expressions. Recent work addresses the scalability limitation, but either restricts the predicate expressiveness \[74\] or assumes a static environment in which the index is constructed offline \[12, 211, 82\]. Our goal is to address scalability, expressiveness, dynamic construction, and adaptation by proposing a self-adjusting index structure that is specifically geared towards high-dimensionality over discrete and finite domains. To achieve these goals, we propose \textbf{BE-Tree}, a tree structure to efficiently index and match large sets of Boolean Expressions defined over an expressive predicate language in a high-dimensional space \[176\]. \textbf{BE-Tree} is dynamically constructed through a two-phase space-cutting (i.e., partitioning and clustering) technique that exploits the discrete and finite structure of both the subscription and event space. Another distinct feature of \textbf{BE-Tree} is a novel self-adjusting mechanism that adapts as subscription and event workloads change.

In this chapter, we make the following contributions:

1. Unify subscription and event language to enable a more expressive matching semantics in Section 3.2, and we evaluate various matching semantics in Section 3.8.

2. Introduce a novel data structure, \textbf{BE-Tree}, that through our proposed two-phase space-cutting supports an extensive set of operators and dynamic expression schemata, gracefully scales to millions of subscriptions, thousands of dimensions, and dozens of predicates per subscription and event in Section 3.3.

3. Present formal analysis of \textbf{BE-Tree}’s key properties in Sections 3.3 and 3.6.

4. Develop a set of novel self-adjusting policies for \textbf{BE-Tree} that continuously adapt to both subscription and event workload changes (Section 3.5),
5. Introduce a novel cache-conscious lazy and bitmap-based Boolean predicate evaluation and a bloom filtering techniques to substantially improve BE-Tree’s matching time (Section 3.7),

6. Present the first comprehensive evaluation framework, including both micro and macro experiments, that benchmarks state-of-the-art matching algorithms, including SCAN [220], SIFT [220], Gryphon [12], our improved Gryphon [12], Access Predicate Pruning (APP) [77], Propagation [74], k-index [211], and GPU-based CLCB [142, 58, 143] (cf. Section 3.8).

3.2 Expression Matching Model

In this section, we formalize our Boolean expression language and data model followed by our stateless matching semantics.

3.2.1 Expression Language

Traditionally, pub/sub matching algorithms take as input a set of subscriptions (conjunction of Boolean predicates) and an event (an assignment of a value to each attribute), and return a subset of subscriptions satisfied by the event. Unlike most existing work, we model both subscriptions and events as Boolean expression. This generalization gives rise to more expressive matching semantics while still encompassing the traditional pub/sub matching problem.

Each Boolean expression is a conjunction of Boolean predicates. A predicate is a triple, consisting of an attribute uniquely representing a dimension in \( n \)-dimensional space, an operator, and a set of values, denoted by \( P_{\text{attr}, \text{opt}, \text{val}}(x) \) or more concisely as \( P(x) \). A predicate either accepts or rejects an input \( x \) such that \( P_{\text{attr}, \text{opt}, \text{val}}(x) : x \rightarrow \{ \text{True}, \text{False} \} \), where \( x \in \text{Dom}(P_{\text{attr}}) \) and \( P_{\text{attr}} \) is the predicate’s attribute. Formally, a Boolean expression \( \Omega \) is defined over an \( n \)-dimensional space as follows:

\[
\Omega = \{ P_{1{\text{attr}}, \text{opt}, \text{val}}(x) \land \cdots \land P_{k{\text{attr}}, \text{opt}, \text{val}}(x) \},
\]

where \( k \leq n \); \( i, j \leq k \), \( P_{i{\text{attr}}} = P_{j{\text{attr}}} \iff i = j \). \hfill (3.1)

We support an expressive set of operators for the most common data types: relational operators (<, \( \leq \), =, \( \neq \), \( \geq \), >), set operators (\( \in \), \( \notin \)), and the SQL BETWEEN operator.

3.2.2 Matching Semantics

Our expression subscription and event language enables a wide range of matching semantics, including stabbing subscription, stabbing event, symmetric matching, containment matching, enclosure matching, and exact matching.

We start with the classical pub/sub matching problem: Given an event \( \omega \) and a set of subscriptions, find all subscriptions \( \Omega_i \in \Omega \) satisfied by \( \omega \). We refer to this problem as stabbing subscription\(^1\) SS(\( \omega \)), and specify the problem as follows:

\(^1\)This is a generalization of stabbing query, which determines which of a collection of intervals overlap a query point.
Chapter 3. Query Indexing (Main Memory)

27

symmetric matching to it as and a set of subscriptions, find all subscriptions \( \Omega \) symmetric pub/sub paradigm [172]. This bidirectional matching problem is defined as given an event \( \omega \), find all subscriptions \( \Omega \) satisfying \( \omega \), stabbing event \( \text{SE}(\omega) \), and it is given as

\[
\text{SE}(\omega) = \{ \Omega_i | \forall \text{attr} \text{.val} \in \omega, \exists \text{attr} \text{.val} \in \Omega_i, \text{attr} = \text{attr}, \exists x \in \text{Dom}(\text{attr}), P_\omega(x) \land P_o(x) \}\.
\]

The reverse direction of stabbing subscription is defined as given an event \( \omega \) and a set of subscriptions, find all subscriptions \( \Omega_i \in \Omega \) satisfying \( \omega \), stabbing event \( \text{SE}(\omega) \), and it is given as

\[
\text{SE}(\omega) = \{ \Omega_i | \forall \text{attr} \text{.val} \in \omega, \exists \text{attr} \text{.val} \in \Omega_i, \text{attr} = \text{attr}, \exists x \in \text{Dom}(\text{attr}), P_\omega(x) \land P_o(x) \}\.
\]

The stabbing event enables us to formalize symmetric stabbing which is necessary to model the symmetric pub/sub paradigm [172]. This bidirectional matching problem is defined as given an event \( \omega \) and a set of subscriptions, find all subscriptions \( \Omega_i \in \Omega \) satisfied by \( \omega \) and satisfying \( \Omega \), which we refer to it as symmetric matching \( \text{SM}(\omega) \)

\[
\text{SM}(\omega) = \{ \text{SS}(\omega) \cap \text{SE}(\omega) \}
\]

We can also answer much stronger matching semantics, given an event \( \omega \) and a set of subscriptions, find all subscriptions \( \Omega_i \in \Omega \) enclosed by \( \omega \), denoted by containment matching \( \text{CM}(\omega) \)

\[
\text{CM}(\omega) = \{ \Omega_i | \forall \text{attr} \text{.val} \in \omega, \exists \text{attr} \text{.val} \in \Omega_i, \text{attr} = \text{attr}, \forall x \in \text{Dom}(\text{attr}), P_\omega(x) \land P_o(x) \}\.
\]

The reverse direction of containment matching is defined as given an event \( \omega \) and a set of subscriptions, find all subscriptions \( \Omega_i \in \Omega \) enclosing \( \omega \), enclosure matching \( \text{EM}(\omega) \)

\[
\text{EM}(\omega) = \{ \Omega_i | \forall \text{attr} \text{.val} \in \omega, \exists \text{attr} \text{.val} \in \Omega_i, \text{attr} = \text{attr}, \forall x \in \text{Dom}(\text{attr}), P_\omega(x) \land P_o(x) \}\.
\]

Lastly, we define exact matching \( \text{XM}(\omega) \): given an event \( \omega \) and a set of subscriptions, find all subscriptions \( \Omega_i \in \Omega \) enclosed by \( \omega \) and enclosing \( \omega \)

\[
\text{XM}(\omega) = \{ \text{CM}(\omega) \cap \text{EM}(\omega) \}\.
\]

The matching semantics supported by BE-Tree can be summarized as follows. BE-Tree returns an approximate answer (a subset of answer) for stabbing event and enclosure matching, yet it returns an exact answer for all matching problems of immediate practical interests in Boolean expression indexing: stabbing subscription, symmetric matching, containment matching, and exact matching. Furthermore, our matching semantics can further be classified as either forward matching (traditional database indexing semantics, e.g., R-Tree), reverse matching (traditional pub/sub matching semantics, in which the role of query and objects is reversed), or bi-directional matching (symmetric matching). The subtle
difference between forward and reverse matching is due to the fact that the database indexing semantics differs with our proposed semantics in an important respect. Our reverse matching semantics solves the reverse database matching problem. In the database context, querying (matching) means finding the relevant tuples (events) for a given query (subscription). But in our context, matching (querying) means finding the relevant subscriptions (queries) for a given event (tuple).

The three categories of the matching semantics are (1) forward matching, namely, stabbing event and enclosure matching; (2) reverse matching, namely, stabbing subscription and containment matching; and (3) bi-directional matching, namely, symmetric matching and exact matching. Alternately, BE-Tree can be characterized as to return exact answers for both reverse matching and bi-directional matching and to return an approximate answers for forward matching. The distinction between forward and reverse matching semantics is yet another design principle that sets apart BE-Tree from traditional R-Tree family of indexes [87]).

In the remainder of this chapter, we simply refer to a Boolean expression as an “expression”, and we use the term “expression” also to refer to both a subscription and an event. Without loss of generality, whenever it is not clear from the context, we use the term subscription and event to distinguish between a set of expressions stored in the index and input expression to be matched, respectively.

3.3 BE-Tree Organization

BE-Tree dynamically indexes large sets of expressions (i.e., subscriptions) and efficiently determines which of these expressions match an input expression (i.e., event). BE-Tree supports Boolean expressions with an expressive set of operators defined over a high-dimensional space. The main challenge in indexing a high-dimensional space is to effectively cut the space in order to prune the search at lookup time. BE-Tree copes with this challenge—the curse of dimensionality—through a two-phase space-cutting technique that significantly reduces the complexity and the level of uncertainty of choosing an effective criterion to recursively cut the space and to identify highly dense subspaces. The two-phases BE-Tree employs are: (1) space partitioning which is the global structuring to determine the best splitting attribute $\text{attr}_i$, i.e., the $i$th dimension (Section 3.3.2) and (2) space clustering which is the local structuring for each partition to determine the best grouping of expressions with respect to the expressions’ range of values for $\text{attr}_i$ (Section 3.3.3).

This two-phase approach, the space partitioning followed by the space clustering, introduces new challenges such as how to determine the right balance between the space partitioning and clustering, and how to develop a robust principle to alternate between both. These new challenges are addressed in BE-Tree by exploiting the underlying discrete and finite domain properties of the space. We begin by discussing the structure and the dynamics of BE-Tree before presenting the main design principles behind BE-Tree. All these are prerequisites to the actual, but much simpler, expression matching with BE-Tree, described in Section 3.4.

3.3.1 BE-Tree Structure

BE-Tree is an $n$-ary tree structure in which a leaf node contains a set of expressions and an internal node contains partial predicate information (e.g., an attribute and a range of values) about the expressions in its descendant leaf nodes. We distinguish among three classes of nodes: a partition node ($p$-node) which maintains the space partitioning information (an attribute), a cluster node ($c$-node) which maintains
the space clustering information (a range of values), and a leaf node \( l \)-node which stores the actual expressions. Moreover, \( p \)-nodes and \( c \)-nodes are organized in a special directory structure for fast space pruning. Thus, a set of \( p \)-nodes is organized in a partition directory (\( p \)-directory), and a set of \( c \)-nodes is organized in a cluster directory (\( c \)-directory). Before giving a detailed account of each node type and the BE-Tree dynamics, we outline the structural properties of BE-Tree as shown in Figure 3.1, and give an example showing the overall dynamics of BE-Tree.

**Example** Initially, BE-Tree has an empty root node which consists of a \( c \)-node that points only to an \( l \)-node. Upon arrival, new expressions (subscriptions) are inserted into the root’s \( l \)-node, and once the size of the \( l \)-node exceeds the leaf capacity—a tunable system parameter—the space partitioning phase is triggered and a new \( \text{attr}_i \) for splitting the \( l \)-node is chosen. The new \( \text{attr}_i \) results in the creation of a new \( p \)-node. The \( \text{attr}_i \) is chosen based on statistics gathered from expressions (subscriptions) in the overflowing \( l \)-node. The selected attribute is passed on to the space clustering phase that divides the domain of the \( \text{attr}_i \) into a set of intervals, in which each range of values is assigned to a new \( c \)-node, and all the expressions having a predicate on \( \text{attr}_i \), in the overflowing \( l \)-node, are distributed across these newly created \( c \)-nodes based on the \( c \)-nodes’ range of permitted values. In brief, BE-Tree recursively partitions and clusters the space. These two phases together recursively identify and refine dense subspaces, in order to maintain the size of each \( l \)-node below a threshold.

Strictly speaking, in BE-Tree, each \( p \)-node is assigned an \( \text{attr}_i \) such that all the expressions in its descendant \( l \)-nodes must have a predicate defined over \( \text{attr}_i \). Similarly, each \( c \)-node is associated with a predicate \( P_i^{\text{attr}, \text{opt}, \text{val}}(x) \) (i.e., a range of permitted values), and all the expressions in its descendant \( l \)-nodes must have a predicate \( P_j^{\text{attr}, \text{opt}, \text{val}}(x) \) such that

\[
(P_i^{\text{attr}} = P_j^{\text{attr}}) \land (\forall x \in \text{Dom}(P_j^{\text{attr}}), P_j(x) \rightarrow P_i(x)).
\]  

Provided that each \( c \)-node is denoted by a predicate \( P_j(x) \), we can assign each \( l \)-node a key \( \text{key}_j \) defined as a conjunction of all \( c \)-nodes’ predicate along the path from the root to the \( l_j \)-node.

In what follows, in order to uniquely identify the same category of nodes, a unique id is assigned to each node. For example, in order to refer to the \( j \)-th \( l \)-node, we write \( l_j \)-node. In addition, we refer to BE-Tree internal parameters as follows: \( \text{max}_{\text{sap}} \) (system max leaf capacity), \( \text{min}_{\text{size}} \) (system min partition size), \( \text{max}_{\text{cap}} \) (\( l_j \)-node max capacity), \( \text{freq}_{\text{window}} \) (update frequency window), \( \theta \) (\( l \)-node recycling threshold), \( \text{ratio}_{\text{exp}} \) (exploit vs. explore ratio), and \( \text{ratio}_{\text{ins}} \) (re-insertion ratio).

### 3.3.2 Space Partitioning

In BE-Tree, space partitioning, conceptually a global adjusting mechanism, is the first phase of our space-cutting technique. The space partitioning is triggered after an \( l_j \)-node overflows and uses a scoring function (cf. Section 3.5) to rank each candidate \( \text{attr}_i \) in order to determine the best attribute for partitioning. Thus, the highest ranking attribute, appearing in at least \( \text{min}_{\text{size}} \) number of expressions, implying that the attribute has a sufficient discriminating power, is chosen for the space partitioning phase. Essentially, this process identifies the next highest ranking dimension, only as the need arises, to segregate expressions into smaller groups based on a high-ranking attribute in order to prune the search space more effectively while coping with the curse of dimensionality.

Upon successful selection of an \( \text{attr}_i \) for space partitioning, a new \( p \)-node for \( \text{attr}_i \) is added to the parent of the overflowing \( l_j \)-node, and the set of expressions in the \( l_j \)-node is divided based on whether
or not they have a predicate defined on $\text{attr}_i$. The partitioning procedure is repeatedly applied to the $l_j$-node to keep its size below $\max_l^{j\text{cap}}$.

The need for $\min_{\text{size}}$ is to avoid ineffective partitioning. For instance, if none of the expressions in an $l_j$-node have a predicate on a common attribute, then there is no computational incentive to form a partition. Therefore, a natural problem that might arise in the space partitioning for any given $l_j$-node is the handling of scenarios for which no candidate attribute exists with size larger than $\min_{\text{size}}$. For such cases, we introduce the notion of an extended $l_j$-node in which the size of the $l_j$-node, $\max_l^{j\text{cap}}$, is increased by a constant factor $\max_l^{\text{cap}}$ such that $\max_l^{j\text{cap}} = \max_l^{j\text{cap}} + \max_l^{\text{cap}}$. Thus, in order to support dynamic expansion and contraction of the leaf node size, after every successful partitioning, the $l_i$-node capacity (to dynamically adjust the leaf node size) is re-evaluated as follows:

$$\max_l^{j\text{cap}} = \begin{cases} \left\lceil \frac{|l_j\text{-node}|}{\max_l^{\text{cap}}} \right\rceil \times \max_l^{\text{cap}}, & \text{if } |l_j\text{-node}| > 0 \\ \max_l^{\text{cap}}, & \text{otherwise} \end{cases}$$  \hspace{1cm} (3.9)

where $|l_j\text{-node}|$ is the size of $l_j$-node.

Another subtle point in the space partitioning is how to guide the attribute selection such that (1) it guarantees that subsequent space partitioning on lower levels of BE-Tree do not ineffectively cycle over a single $\text{attr}_i$; (2) it enables dynamic insertions and deletions without performance deterioration. To achieve these properties, we must ensure that in any path from the root to a leaf node, each $\text{attr}_i$ is selected at most once and that a deterministic clustering is employed after each partitioning (cf. Section 3.3.3). Moreover, we show in Section 3.3.3, the attribute selection restriction is not a limitation; in fact, we prove that it is sufficient to pick each $\text{attr}_i$ at most once yet fully exploiting the domain of $\text{attr}_i$ because when $\text{attr}_i$ is selected, then reselecting it at a lower level of the tree provides no additional benefit.

Moreover, it is evident that the number of partitions for each $c$-node grows linearly in the dimensionality of space. Since each edge from a $c$-node leading to a $p$-node is uniquely identified by a single
attribute, we can employ a hash table over all edges leaving the node, potentially scaling BE-Tree to thousands of dimensions. The inner working of the partition directory is shown in Figure 3.2a.

Finally, as the split operator is required to eliminate overflowing \( l \)-nodes, similarly, a merge operator is necessary to eliminate underflowing \( l \)-nodes. If an \( l \)-node is underflowing, then its contents are either merged with its grandparent’s \( l \)-node or reinserted into BE-Tree. The latter approach is preferred because it provides an opportunity to avoid deterioration of BE-Tree. In any case, if an \( l \)-node is empty and its \( c \)-node has no other children, then the \( l \)-node is removed. This node removal naturally propagates upward removing any non-leaf nodes with no outgoing edges.

### 3.3.3 Space Clustering

Our proposed space partitioning reduces the problem of high-dimensional indexing into one-dimensional interval indexing. Interval indexing is addressed in our space clustering phase, conceptually a local adjusting mechanism. The key insight of the space clustering, and ultimately of BE-Tree, is a deterministic clustering policy to group overlapping expressions (into regions) and a deterministic policy to alternate between the space partitioning and the space clustering. The absence of a predictable policy gives rise to the dilemma of whether to further pursue the space clustering or to switch back to the space partitioning. Besides, once a region is partitioned, that region can no longer be split without running into the cascading split problem [84]: an unpredictable chain reaction that propagates downwards, potentially effecting each node at every level of the tree including the leaf nodes. Thus, a deterministic clustering policy that is influenced by the insertion sequence is either prone to ineffective regions that do not take advantage of the dimension selectivity to effectively prune the search space or prone to suffer from substantial performance overhead due to the cascading split problem. Therefore, to achieve determinism in our space clustering, while supporting dynamic insertion and deletion, our structure must be independent of insertion sequence.

To address these challenges, we propose a grid-based approach, with unique splitting and merging
policies, to build the clustering directory in \texttt{BE-Tree}. The clustering directory is a hierarchical structure that organizes the space into sets of expressions by recursively cutting the space in halves. A key feature of this grid-based clustering is a forced split rule that (1) avoids the cascading split problem and that (2) enables deterministic clustering and partition-clustering alteration strategies that are independent of the insertion sequence. To describe the dynamics of our clustering directory, first, we formally define a few concepts.

**Definition 3.** A bucket represents an interval boundary (range of values) over \texttt{attr}_i.

An expression is assigned to a bucket over \texttt{attr}_i only if the set of values defined by the expression’s predicate on \texttt{attr}_i is covered by that bucket; for brevity, we say an expression is assigned to a bucket if the expression is enclosed by that bucket. Furthermore, a bucket has a minimal interval boundary which is a best-effort-smallest interval that encloses all of its expressions. Each bucket is associated with exactly one \texttt{c-node} in \texttt{BE-Tree}, which is responsible for storing and maintaining information about the bucket’s assigned expressions. We further distinguish among four types of buckets.

**Definition 4.** An open bucket is a bucket with a not yet partitioned \texttt{c-node}.

**Definition 5.** A leaf bucket is a bucket that has no children (a bucket that has not been split).

**Definition 6.** A discrete bucket is an atomic bucket that cannot further be split. A discrete bucket is also a leaf bucket, but the reverse direction is not necessarily true.

**Definition 7.** A home bucket is the smallest possible bucket that encloses an expression.

Essentially the clustering directory is constructed based on the following three rules to avoid the cascading split problem and to achieve the deterministic properties of \texttt{BE-Tree}:

**Rule 1.** An expression is always inserted into the smallest bucket that encloses it (insertion rule).

**Rule 2.** A non-discrete bucket is always split before its \texttt{c-node} switches to the space partitioning (forced split rule).

**Rule 3.** An underflowing leaf bucket is merged with its parent only if the parent is an open bucket (merge rule).

The cascading split problem is avoided, first, due to the forced split rule because a bucket is always split before it is partitioned and, second, due to the insertion and merge rules because both current and future expressions are always placed in the smallest bucket that encloses them. Therefore, the partitioned \texttt{c-node} always remains the home bucket to all of its expressions. As a result, there is no benefit or need to further split a bucket that is the home to all of its expressions. This home bucket’s uniqueness property is achieved through a rigorous splitting policy that deterministically cuts the space in half independent of the insertion sequence such that each expression could uniquely be associated to a \texttt{home bucket}.

The deterministic clustering is achieved through a grid-based organization of space in which each overflowing bucket is split in halves. The deterministic partition-clustering alteration is also achieved through the forced split rule which always enforces the split of non-atomic buckets before initiating the space partitioning. Thus, the space partitioning is always applied to expressions that are residing in
### Table 3.1: Operator Transformation

<table>
<thead>
<tr>
<th>Operator</th>
<th>Interval-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i &lt; v_1 )</td>
<td>([v_{\text{min}}, v_1 - 1])</td>
</tr>
<tr>
<td>( i \leq v_1 )</td>
<td>([v_{\text{min}}, v_1])</td>
</tr>
<tr>
<td>( i = v_1 )</td>
<td>([v_1, v_1])</td>
</tr>
<tr>
<td>( i \neq v_1 )</td>
<td>([v_{\text{min}}, v_{\text{max}}])</td>
</tr>
<tr>
<td>( i &gt; v_1 )</td>
<td>([v_1 + 1, v_{\text{max}}])</td>
</tr>
<tr>
<td>( i \geq v_1 )</td>
<td>([v_1, v_{\text{max}}])</td>
</tr>
<tr>
<td>( i \in {v_1, \ldots, v_k} )</td>
<td>([v_1, v_k])</td>
</tr>
<tr>
<td>( i \notin {v_1, \ldots, v_k} )</td>
<td>([v_{\text{min}}, v_{\text{max}}])</td>
</tr>
<tr>
<td>( i \text{ BETWEEN } v_1, v_2 )</td>
<td>([v_1, v_2])</td>
</tr>
</tbody>
</table>

their home bucket such that if the space clustering is further pursed, these expressions are unaffected by it.

In short, the deterministic and no split cascading properties of \emph{BE-Tree} are obtained by satisfying the below specified \emph{BE-Tree} invariance. A complete proof of \emph{BE-Tree}'s invariance is presented in Section 3.6.1. Most importantly, these desired properties are possible only due to the existence of an atomic bucket, which itself is possible only due to the underlying discrete domain property.

**Invariance.** Every expression always resides in the smallest bucket that encloses it, and the \( c \)-node of a non-atomic leaf bucket is never partitioned.

Therefore, the key result with respect to \emph{BE-Tree}'s invariance property, which is proven in Section 3.6.1, can be summarized as follows.

**Theorem 1.** \emph{BE-Tree}'s two-phase space-cutting (space partitioning and clustering) is always safe, i.e., no cascading split problem occurs, and always satisfies the \emph{BE-Tree} invariance.

Subsequently, we present our unique clustering directory structure. We also propose a predicate transformation, which converts our expressive set of operators into an interval boundary that is geometrically compatible with the clustering directory. Finally, we present a set of directory optimization including a specialized clustering and a hybrid clustering directory. The specialized clustering targets a restricted set of operators, and the hybrid clustering utilizes both the generic and the specialized clustering directories to further improve the matching time.

### 3.3.4 Space Clustering Structures

**Space Clustering Structure** The main guiding principles of the \emph{BE-Tree} space clustering are the insertion and the forced split rules. Both are used to dynamically construct \emph{BE-Tree} as follows. The clustering directory starts with an empty top-level bucket which spans the entire domain for \( \text{attr}_i \). Once the leaf node associated to the top-level bucket overflows, the bucket is split in half resulting in the creation of two new child buckets; each child bucket is also assigned a new \( c \)-node and \( l \)-node. Subsequently, expressions in the overflowed leaf node that are enclosed by either of the two child buckets are moved accordingly. This process is recursively applied until either an atomic bucket is reached or every bucket’s \( c \)-node has its \( l_j \)-node below \( \text{max}_{\text{cap}}^j \). Finally, an overflowing non-leaf or an atomic bucket is handled by switching to the partitioning mode. A snapshot of the clustering directory is shown in Figure 3.2b.
In summary, the two-phase space-cutting begins with the space partitioning followed by a sequence of space clusterings until a safe point is reached, i.e., a non-leaf or an atomic bucket, at which point a new instance of the two-phase space-cutting begins starting with the space partitioning. Also, as explained in Section 3.3.2, each attribute is selected at most once in any path along the root of BE-Tree to a leaf node because switching back to the space partitioning occurs only at a safe point, in which the overflowing leaf node (subjected to partitioning) is associated with a home bucket. Therefore, no further clustering is applied because results no additional benefit. In other words, once an attribute is chosen, before switching to the partitioning mode, the space clustering strategy will exploit the entire space to fully leverage the dimension selectivity independent of the insertion sequence.

The clustering directory supports indexing one-dimensional intervals. However, our predicate language supports a richer set of predicates. In Table 3.1, we show the conversion of predicates with different types of operators into one-dimensional intervals, where \( v_{\text{min}} \) and \( v_{\text{max}} \) are the smallest and the largest possible values in the domain, and \( \{v_1, \cdots, v_k\} \) is sorted in ascending order. All predicates transformations are simple algebraic conversions except for the \((\neq, \notin, \in)\) operators. A predicate \( P_j^{(\text{attr}_i, \neq, v_1)}(x) \) implies that every value in the domain of \( \text{attr}_i \) is acceptable except for \( v_1 \). Therefore, under the uniform distribution assumption, the predicate \( P_j(x) \) is satisfied with a high probability by an event expression having a predicate on \( \text{attr}_i \). This observation supports the transformation of the \( \neq \) operator into a one-dimensional interval \([v_{\text{min}}, v_{\text{max}}]\). This transformation results in an early pruning of expressions with a predicate on \( \text{attr}_i \) during the matching of an event that does not have any predicate defined on \( \text{attr}_i \). This pruning strategy is especially effective because the number of predicates per expression is on the order of tens while the number of space dimensions is on the order of thousands. Likewise, the \( \notin \) operator is a generalization of the \( \neq \) operator, which filters out a set of values from the domain instead of a single value. Thus, by converting \( \notin \) to an interval that spans the entire domain, again, we are imposing a filtering strategy to effectively reduce the search space. Finally, applying a similar conversion to the \( \in \) operator results in a further improved filtering strategy compared with the \((\neq, \notin)\) conversion because, now, the assigned interval is bounded by the minimum and the maximum values in the predicate with the \( \in \) operator.

**Specialized Space Clustering Structure** The key observation with regard to the clustering directory is that if the predicate language is restricted to only \((=, \neq, \notin)\) operators, then the clustering directory can be replaced with a hash table and a single bucket that spans the entire domain range, as shown in Figure 3.3a. The hash table and the single filtering bucket trivially satisfy the invariance and the cluster directory rules because every bucket in the hash table is a discrete bucket implying that every bucket is always the home bucket for all the expressions that it is hosting. Similarly, the single bucket stores all the expressions that are transformed into an expression that spans the entire domain, also making this bucket a home bucket for all of its expressions. Hence, the space partitioning in the 2-layer hash-based cluster directory also satisfies the BE-Tree invariance. In addition, the height of 2-layer clustering directory is always two. Hence, the height of BE-Tree is at most \( 2k \), i.e., \( O(k) \), where \( k \) is the maximum number of predicates in an expression.

**Hybrid Space Clustering Structure** In this approach, we combine the idea of the generic and the specialized clustering structures. Therefore, if the expression contains an equality predicate, then it is pushed into a hashtable (specialized structure); otherwise, it is pushed to the generic clustering structure, Figure 3.3b.
Chapter 3. Query Indexing (Main Memory)

3.4 BE-Tree Implementation

Next, we provide an in-depth explanation, together with pseudocode, for the three main operations of BE-Tree, namely, matching, insertion, deletion. Furthermore, for the ease of presentation, without the loss of generality, we focus on matching with stabbing subscription semantics in which event expressions consist of only equality predicates (=).

3.4.1 Matching Pseudocode

Event matching consists of two routines: (1) MatchBETree (Algorithm 1) which checks subscriptions in a leaf node and traverses through BE-Tree’s p-directory and (2) SearchCDir (Algorithm 2) which traverses through BE-Tree’s c-directory.

MatchBETree algorithm takes as inputs: an event, a c-node (BE-Tree’s root initially), and a list to store the matched subscriptions. The algorithm, first, checks all subscriptions in the c-node’s leaf to find the matching subscriptions (Line 1). Second, for every attr, in the event’s predicates, it searches the c-node’s p-directory (Line 4). Lastly, the algorithm calls SearchCDir on all relevant p-nodes (Line 6).

Algorithm 1 MatchBETree(event, cnode, matchedSub)

1: matchedSub ← CheckSub(cnode.lnode)  
   /* Iterate through event’s predicates */
2: for i ← 1 to NumOfPred(event) do  
3:   attr ← event.pred[i].attr  
   /* Check the c-node’s p-directory (hashtable) for attr */
4:   pnode ← SearchPDir(attr, cnode.pdir)  
   /* If attr exists in the p-directory */
5:   if pnode ≠ NULL then  
6:     SearchCDir(event, pnode.cdir, matchedSub)
Algorithm 2 SearchCDir(event, cd, matchedSub)
1: MatchBETree(event, cd.cnode, matchedSub)
2: if IsEnclosed(event, cd.lChild) then
3:   SearchCDir(event, cd.lChild, matchedSub)
4: else if IsEnclosed(event, cd.rChild) then
5:   SearchCDir(event, cd.rChild, matchedSub)

SearchCDir takes as inputs: an event, c-directory, and a list to store the matched subscriptions. The algorithm is as follows: it calls MatchBETree on the c-node of the current c-directory (Line 1), and it recursively calls SearchCDir on the bucket’s left child if the left child encloses the event (Line 3) and on the bucket’s right child if the right child encloses the event (Line 5). In order to support more expressive matching semantics and predicate operators only the IsEnclosed function (in Algorithm 2) algorithm must be changed accordingly.

3.4.2 Insertion Pseudocode

Unlike matching, insertion is rather involved because it also manages the overall dynamics of BE-Tree, i.e., the space partitioning and the space clustering. To insert, BE-Tree’s root and a subscription is passed to the InsertBETree (Algorithm 3) that attempts to find an l-node with the highest score that encloses the subscription. Essentially the insertion is done recursively in two stages. Initially, the p-directory is searched for every unused attr, in the subscription, and the attr\textsubscript{max} with highest p-node score is selected (Lines 2-11); an unused attr\textsubscript{i} is one that has not been selected at a higher level of BE-Tree by the InsertBETree. Subsequently, if no such attr\textsubscript{max} is found, then the subscription is inserted into the l-node of the current c-node (Line 13). However, if an attr\textsubscript{max} is found, then the subscription is pushed down to its corresponding p-node (Line 19).

On the one hand, when attr\textsubscript{max} is found (Line 19), the c-directory of the corresponding p-node is searched for the smallest possible c-node that encloses the subscription, the search is done through InsertCDir (Algorithm 4). Upon choosing the smallest c-node, the subscription advances to the next level of BE-Tree, and the routine InsertBETree is recursively called on the new c-node. After the recursive call, the function UpdatePartitionScore (Line 21) is invoked, which implements our proposed cost-based ranking function (cf. Section 3.5) based on Equation 3.10 or its simpler form given by Equation 3.14.

On the other hand, when no attr\textsubscript{max} is found (Line 13), the l-node at the current level is declared as the best l-node to hold the new subscription so that InsertBETree’s recursion reaches the base case and terminates; however, after the insertion into the l-node, the node may overflow, which, in turn, triggers BE-Tree’s two-phase space-cutting technique: partitioning and clustering.

In particular, if the chosen l-node is at the root level, in which no partitioning or clustering has yet taken place, then the space partitioning is invoked first (Line 15) because the space clustering is feasible only after the space is partitioned; otherwise, the space clustering is invoked (Line 17).

SpacePartitioning (Algorithm 5) proceeds as follows. It uses a scoring function (e.g., selectivity or popularity) to find an unused attribute with the highest ranking, attr\textsubscript{max}, that appears in predicates of the overflowing subscription set (Line 3); consequently, a new p-node is created for the attr\textsubscript{max}. Next, the algorithm iterates over all the subscriptions in the overflowing l-node, and moves all subscriptions having a predicate defined over attr\textsubscript{max} into the c-directory of the attr\textsubscript{max}’s p-node (Lines 5-8). Lastly, the space clustering is called on the c-directory to resolve any potential overflows resulting from moving
Algorithm 3 InsertBETree(sub, cnode, cdir)
1: /* Find attr with max score not yet used for partitioning */
2: if cnode.pdir ≠ NULL then
3:   for i = 1 to NumOfPred(sub) do
4:     if !IsUsed(sub.pred[i]) then
5:       attr ← sub.pred[i].attr
6:       pnode ← SearchPDir(attr, cnode.pdir)
7:       if pnode ≠ NULL then
8:         foundPartition = true;
9:         if maxScore < pnode.score then
10:          maxPnode ← pnode
11:          maxScore ← pnode.score
12: /* if no partitioning found then insert into the l-node */
13: if !foundPartition then
14:   Insert(sub, cnode.lnode)
15: /* if c-node is the root then partition; otherwise cluster */
16: if isRoot(cnode) then
17:   SpacePartitioning(cnode)
18: else
19:   SpaceClustering(cdir)
20: else
21:   maxCdir ← InsertCDir(sub, maxPnode.cd)
22: InsertBETree(sub, maxCdir.cnode, maxCdir)
23: UpdatePartitionScore(maxPnode)

Algorithm 4 InsertCDir(sub, cdir)
1: if IsLeaf(cd) then
2:   return cd
3: else
4:   if IsEnclosed(sub, cd.lChild) then
5:     return InsertCDir(sub, cd.lChild)
6: else if IsEnclosed(sub, cd.rChild) then
7:     return InsertCDir(sub, cd.rChild)
8: return cd

subscriptions (Line 9); the entire process is repeated until the l-node is no longer overflowing. Lastly, by calling the function UpdateClusterCapacity (Line 10), the cluster capacity is updated based on Equation 3.9.

SpaceClustering (Algorithm 6) is always invoked after the space is partitioned in order to resolve any overflowing l-node by recursively cutting the space in half, and only if the space clustering is unfeasible, then it switches back to the space partitioning. The space clustering is unfeasible when an overflowing l-node is associated to a c-directory bucket that is either a non-leaf bucket, in which further splitting does not reduce the l-node size, or an atomic bucket, in which further splitting is not possible (Line 4). If the space clustering is feasible, then the algorithm splits the current bucket directory in half and moves subscriptions accordingly (Lines 7-13). Next the algorithm recursively calls SpacePartitioning on the current bucket (Line 14) and calls SpaceClustering on the current bucket’s left and right child (Lines 15-16). Lastly, by calling the function UpdateClusterCapacity (Line 17), the cluster capacity is updated based on Equation 3.9.
Algorithm 5 SpacePartitioning(cnode)
1: lnode ← cnode.lnode
2: while IsOverflowed(lnode) do
3:   attr ← GetNextHighestScoreUnusedAttr(lnode)
   /* Create new partition for the next highest score attr */
4:   pnode ← CreatePDir(attr, cnode.pdir)
   /* Move all the subscriptions with predicate on attr */
5:   for sub ∈ lnode do
6:     if sub has attr then
7:       cdir ← InsertCDir(sub, pnode.cdir)
8:       Move(sub, lnode, cdir.cnode.lnode)
9:     SpaceClustering(pnode.cdir)
10:    UpdateClusterCapacity(lnode)

Algorithm 6 SpaceClustering(cdir)
1: lnode ← cdir.cnode.lnode
2: if !IsOverflowed(lnode) then
3:   return
4: if !IsLeaf(cdir) or IsAtomic(cdir) then
5:   SpacePartitioning(cdir.cnode)
6: else
7:   cdir.lChild ← [cdir.startBound, cdir.endBound/2]
8:   cdir.rChild ← [cdir.endBound/2, cdir.endBound]
9:   for sub ∈ lnode do
10:      if IsEnclosed(sub, cdir.lChild) then
11:        Move(sub, lnode, cdir.lChild.cnode.lnode)
12:      else if IsEnclosed(sub, cdir.rChild) then
13:        Move(sub, lnode, cdir.rChild.cnode.lnode)
14:     SpacePartitioning(cdir.cnode)
15:    SpaceClustering(cdir.lChild)
16:    SpaceClustering(cdir.rChild)
17:   UpdateClusterCapacity(lnode)

3.4.3 Deletion Pseudocode

The algorithm for deleting subscription consists of two routines: (1) DeleteBETree (Algorithm 7) which checks all subscriptions in a leaf node and traverses through BE-Tree’s p-directory and (2) SearchDeleteCDir (Algorithm 8) which traverses through BE-Tree’s c-directory.

The DeleteBETree algorithm takes as inputs: a unique subscription to be removed, a c-node (BE-Tree’s root initially). The algorithm, first, checks all subscriptions in the c-node’s leaf to find the subscription to be removed. (Line 1). Second, for every attr, in the subscription’s predicates, it searches the c-node’s p-directory (Line 6). Next, the algorithm calls SearchDeleteCDir on all relevant p-nodes (Line 8). Now at any point, if the subscription is found (Line 9), the search for the subscription is stopped. Once the subscription is found and removed, then the algorithm enters the garbage collection mode and begins removing any empty nodes and directories (Lines 11-19). First, it removes any empty p-node that held the deleted subscription (Line 13).

After removing the p-node, if the p-directory becomes empty, then the p-directory is also removed (Line 15). Next if the l-node of the current c-node is empty, then the l-node is also removed (Line 17). Finally, once l-node and p-directory for the current c-node are removed, the c-node itself is also removed.
Algorithm 7 DeleteBETree\((sub, cnode)\)

1: \(isFound \leftarrow deleteSubFromLeaf(cnode.lnode)\)
2: if \(!isFound\) then
3:  /* Iterate through subscription’s predicates */
4:  for \(i \leftarrow 1\) to \(\text{NumOfPred}(sub)\) do
5:    \(attr \leftarrow sub.pred[i].attr\)
6:    /* Check the \(c\)-node’s \(p\)-directory (hashtable) for \(attr\) */
7:    \(pnode \leftarrow \text{SearchPDir}(attr, cnode.pdir)\)
8:    /* If \(attr\) exists in the \(p\)-directory */
9:    if \(pnode \neq \text{NULL}\) then
10:      \(isFound \leftarrow \text{SearchDeleteCDir}(sub, pnode.cdir)\)
11:      if \(isFound\) then
12:       break
13:    if \(isFound\) then
14:      if \(\text{isEmpty}(pnode)\) then
15:        removePNode\((pnode)\)
16:      if \(\text{isEmpty}(cnode.pdir)\) then
17:        removePDir\((cnode.pdir)\)
18:      if \(\text{isEmpty}(cnode.lnode)\) then
19:        removeLNode\((cnode.lnode)\)
20:      if \(\text{isEmpty}(cnode)\) then
21:        removeCNode\((cnode)\)
22: return \(isFound\)

(Line 19). Notably, there is no difference between removing a \(c\)-node and a \(p\)-node; once a node becomes empty (having no descents), then it is simply removed.

Algorithm 8 SearchDeleteCDir\((sub, cdir)\)

1: if \(\text{IsEnclosed}(event, cdir.lChild)\) then
2:  \(isFound \leftarrow \text{SearchDeleteCDir}(sub, cdir.lChild)\)
3: else if \(\text{IsEnclosed}(event, cdir.rChild)\) then
4:  \(isFound \leftarrow \text{SearchDeleteCDir}(sub, cdir.rChild)\)
5: else
6:  \(isFound \leftarrow \text{DeleteBETree}(sub, cdir.cnode)\)
7: if \(isFound\) then
8:  if \(\text{isEmpty}(cdir.lChild)\) then
9:    removeBucket\((cdir.lChild)\)
10:  if \(\text{isEmpty}(cdir.rChild)\) then
11:    removeBucket\((cdir.rChild)\)
12:  if \(\text{isEmpty}(cdir)\) then
13:    removeCDir\((cdir)\)
14: return \(isFound\)

Algorithm \text{SearchDeleteCDir} takes as inputs: a unique subscription to be removed, and a \(c\)-directory. The algorithm is as follows: it recursively calls \text{SearchDeleteCDir} on the bucket’s left child if the left child encloses the subscription (Line 2) or on the bucket’s right child if the right child encloses the subscription (Line 4). If neither left nor right buckets enclose the subscription, then the algorithm calls \text{DeleteBETree} on the \(c\)-node of the current \(c\)-directory (Line 6). Once the subscription to be removed is found, then any empty left or right buckets and \(c\)-directory are removed (Lines 7-13).
3.4.4 Concrete Example

Next we present an example to further elucidate the insertion algorithm. The final BE-Tree is shown in Fig 3.4. Initially, BE-Tree is empty. After inserting $S_1 - S_4$, the root’s l-node overflows, and based on $\text{InsertBETree}$ (Line 15), the root is partitioned, and the attribute $[a]$ is selected as $\text{attr}_{\text{max}}$, and $S_1$ and $S_4$ are pushed down to the next level of BE-Tree. After inserting $S_5 - S_6$, another overflow occurs at the root, which results in selecting $[d]$ as $\text{attr}_{\text{max}}$, and, similarly, $S_3$, $S_5$, and $S_6$ are pushed down the tree. After inserting $S_7 - S_8$, both having predicates defined on $[a]$, they are directed toward the l-node containing $S_1$ and $S_4$ along the root’s p-node with value $[a]$. Consequently, this node will overflow, and based on $\text{InsertBETree}$ (Line 17), the l-node is clustered. Finally, the insertion of $S_9$ will overflow the top-level (non-leaf) c-directory, reachable through the root’s p-node with value $[a]$; thereby, triggering the space partitioning and selecting $[b]$ as the next $\text{attr}_{\text{max}}$ through $\text{SpaceClustering}$ (Line 5).

3.5 BE-Tree Self-adjustment

BE-Tree self-adjustment is based on a cost-based ranking function and adaptation policies that utilize this ranking function.

3.5.1 Cost-based Ranking Function

BE-Tree’s ranking objective directly reduces the matching cost as opposed to a ranking that is founded solely on popularity measures and, consequently, biased towards either the least or the most popular key [74], which is a deviation from the actual index objective. Our BE-Tree’s ranking objective, which directly reduces the matching cost, is formulated based on the notion of false candidates.
**Definition 8.** False candidates are the expressions (subscriptions) retrieved that are not matched by an input expression (event).

We formalize BE-Tree’s ranking objective based on the matching cost as follows. The matching cost is defined as the total number of predicate evaluations that is broken down into minimizing false candidate computations and minimizing true candidate computations. The false candidate computation is the total number of predicate evaluations until an unsatisfied predicate is reached which discards the prior computations (i.e., discarded predicate evaluation) along the search path, therefore, penalizing multiple search paths of the tree that are discarded eventually. Also, the false candidate computation tracks the total number of predicates evaluated for each unsatisfied expression; therefore, penalizing keys (i.e., access predicate) that produce many false candidates. The true candidate computation is the number of predicate evaluations before reporting a set of expressions as matched, namely, promoting the evaluation of common predicates among a set of expressions exactly once.

We define a ranking model for each node in BE-Tree using the proposed matching cost. For an improved ranking accuracy, we also introduce the notion of covered and subsumed predicates. The covered predicates are defined as all predicates \( P_{l_j}(x) \) in each \( l_j \)-node’s expressions such that there exists a \( P_i(x) \in \text{key}_{l_j} \) and \( \text{attr}_{l_j} = \text{attr}_i \) because by the definition of the leaf node’s key, all the \( P_{l_j}(x) \) must be covered by \( P_i(x) \). However, if \( P_i(x) \) and \( P_l(x) \) are also equivalent, i.e., \( \forall x \in \text{Dom}(\text{attr}_l) \) \( P_i(x) \Leftrightarrow P_l(x) \), then \( P_l(x) \) is considered subsumed and not covered. Thus, subsumed predicates are preferred because the covered predicates are approximate and must be re-evaluated at the leaf level for each expression.

The ranking model assigns a rank to each node \( n_i \) using the function \( \text{Rank}(n_i) \) which is a combination of the \( \text{Loss}(n_i) \) and \( \text{Gain}(l_j) \) functions. \( \text{Loss}(n_i) \) computes for each node the false candidates generated over a window of \( m \) events. \( \text{Gain}(l_j) \) is defined for each \( l_j \)-node, and it is the combination of the number of subsumed and covered predicates for each of its expressions. Formally, \( \text{Rank}(n_i) \), \( \text{Loss}(n_i) \), and \( \text{Gain}(l_j) \) are defined as follows:

\[
\text{Rank}(n_i) = \begin{cases} 
(1 - \alpha)\text{Gain}(n_i) - \alpha\text{Loss}(n_i) & \text{if } n_i \text{ is a } l\text{-node} \\
\left( \sum_{n_j \in \text{des}(n_i)} \text{Rank}(n_j) \right) - \alpha\text{Loss}(n_i) & \text{otherwise},
\end{cases}
\]

(3.10)

where \( 0 \leq \alpha \leq 1 \).

\[
\text{Loss}(n_i) = \sum_{e' \in \text{window}_m(n_i)} \frac{\# \text{discarded pred eval for } e'}{|\text{window}_m(n_i)|}.
\]

(3.11)

\[
\text{Gain}(l_j) = (1 - \beta)\text{Gain}_s(l_j) + \beta\text{Gain}_c(l_j), \quad 0 \leq \beta \leq 1.
\]

(3.12)

\[
\text{Gain}_s(l_j) = \# \text{subsumed pred}, \quad \text{Gain}_c(l_j) = \# \text{covered pred}.
\]

(3.13)

The proposed ranking model is simply generalized for splitting an overflowing node \( l_j \)-node using a new attr\(_i\), and it is given by

\[
\text{Rank}(l_i) = (1 - \alpha)\text{Gain}(l_i) - \alpha\text{Loss}(l_i), \quad \text{where } 0 \leq \alpha \leq 1,
\]

(3.14)
where $Gain(l_i)$ is approximated by the number of expressions that have a predicate on $\text{attr}_i$ and $Loss(l_i)$ is estimated by constructing a histogram in which $Loss(l_i)$ is the average bucket size in the histogram. Essentially, the average bucket size estimates the selectivity of $\text{attr}_i$, meaning, in the worst case the number of false candidates is equal to the average bucket size. Alternatively, in an optimistic approach $Loss(l_i)$ is initially set to zero to eliminate any histogram construction and to rely on the matching feedback mechanism for adjusting the ranking, if necessary. This optimistic approach initially estimates the popularity of $\text{attr}_i$ as opposed to selectivity of $\text{attr}_i$. Based on our experimental evaluation, the optimistic approach results in an improved matching and insertion time.

Similarly, based on empirical evidence, this cost model can be further simplified by setting both parameters $\alpha$ and $\beta$ to 0.5; hence, giving equal weight to the $Gain()$ and $Loss()$ functions and to the $Gain_s()$ and $Gain_c()$ functions. This simplification is possible because in the majority of our synthetic and real experiments the rate of false candidates was sufficiently low such that altering the values of these parameters had negligible influence on the overall BE-Tree matching rate. However, in certain special settings, these two parameters could be invaluable, namely, to deal with fluctuations in event stream or to model the effectiveness of subsumed vs. covered predicates.

In general, the parameter $\alpha$ can play an important role in order to smoothen sudden spikes in the event stream or to adapt to a stream with high fluctuation rates. A low value of $\alpha$ places smaller weight on recent changes captured by our $Loss()$ function and puts heavier weight on the subscription workload characteristics that are captured by our $Gain()$ function. In contrast, a high value of $\alpha$, places larger weight on an event stream and adapts to stream fluctuations. Alternatively, one can view the role of $\alpha$ as adapting to either event workload (a high value of $\alpha$) or subscription workload (a low value of $\alpha$).

Our second parameter $\beta$ is used to establish the effectiveness of subsumed and covered predicates while observing the event stream. Initially, it is assumed that both subsumed and covered predicates are equally effective (i.e., $\beta = 0.5$), meaning that a covered predicate has a low rate of generating false candidates. However, if during the matching process, it is observed that a covered predicate has a much higher rate of generating false candidates, then the parameter $\beta$ is tuned accordingly. For instance, if on average, a covered predicate generates a false candidate at a rate of 80%, then $\beta$ is set to 0.1 in order to favor subsumed predicates accordingly.

$$\beta = 0.5 - \frac{\phi}{2},$$

(3.15)

where $\phi$ is the observed rate of false candidates for covered predicates. The effectiveness (and dynamic tuning) of the $\beta$ parameter is studied in Section 3.8.6.

### 3.5.2 Adaptation Policies

In BE-Tree, three strategies are considered to utilize the cost-based ranking function and to further avoid tree degeneration: recycling $l$-nodes, reinserting Boolean expressions, and exploration vs. exploitation.

**Recycling $l$-nodes** is BE-Tree’s main self-adjusting policy that monitors each node over a frequency window, $freq_{window}$, of the number of insertion, deletion, and matching operations. If at the end of each $freq_{window}$ interval for each node, the rank of a node drops below the threshold, $\theta$, then the entire contents of that node, including all of its descendant leaf nodes (if any), are removed and re-inserted into BE-Tree.

**Reinserting Boolean Expression** is exercised prior to invoking the space clustering and the space
partitioning. Reinsertion is geared towards adapting BE-Tree to changes in the subscription workload. This policy targets leaf nodes by randomly selecting a subset of a leaf node content, driven by the \textit{ratio}_{ins} parameter, and it reinserts the selected elements.

\textbf{Exploration vs. Exploitation} is a self-adjusting policy that is triggered when a new expression is inserted in BE-Tree. At every level of BE-Tree, an attribute from the expression is selected such that with probability \textit{ratio}_{exp} the selected attribute is ranked the highest in the current level (exploitation) otherwise the selected attribute is randomly chosen (exploration). This active policy enables exploring the entire space while benefiting from existing statistics and avoids making decisions based only on past selections.

### 3.6 BE-Tree Theoretical Analysis

In this section, we first present a complete proof of BE-Tree’s invariance followed by BE-Tree space and time complexity analysis of BE-Tree.

#### 3.6.1 BE-Tree Proof of Correctness

We first proof a general property, namely, no deadlock occurs in the insertion process; thus, there always exists a bucket in the clustering directory that can accommodate newly inserted expressions.

**Lemma 1.** A leaf bucket is either an open or a discrete bucket.

**Proof.** There are two main operations: (1) Bucket splitting that is the result of insertions and (2) bucket merging that is the result of deletions. In the case of bucket splitting, Lemma 1 follows simply from the forced split rule because a bucket that is associated with a partitioned \(c\)-node, must have been split once, implying that it cannot be a leaf unless it is a discrete bucket. Therefore, if a leaf bucket is not discrete, then it is always open because it has no children and has not yet been split. In the case of bucket merging, if a bucket is underflowing, then it can be merged with its immediate parent only if the parent is an open bucket resulting, again, in a single open leaf bucket.

**Lemma 2.** A discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting, i.e., the smallest bucket that can enclose them.

**Proof.** Since a discrete bucket is the smallest possible bucket in the clustering directory, it is always the home bucket for its expressions. A non-leaf bucket by definition must have at least two children, and given the insertion rule, it follows that a non-leaf bucket is always the home bucket for its expressions.

**Theorem 2.** The insertion of an expression incurs no deadlock, namely, there exists either the home bucket for the expression or an open leaf bucket that encloses it.

**Proof.** If there exists a home bucket for an expression, then the correctness is trivial because the expression is simply inserted into the home bucket, and there is no deadlock. However, if the home bucket does not exist, then we must show that there exists at least an open bucket that encloses it. In fact, a much stronger claim can be made which is that if the home bucket does not exist, then there is exactly one open leaf bucket, a unique non-discrete leaf bucket, that encloses it. If the home bucket does not exist, then there must be exactly one leaf bucket that encloses its home bucket due to recursively splitting the space in half. Since the leaf bucket encloses the expression’s home bucket, it is not a discrete
bucket. Therefore, based on Lemma 1, the leaf bucket must be an open leaf bucket. Hence, no deadlock exists.

Next, we define the notions of safe partitioning and clustering, i.e., avoiding the cascading split problem, followed by a proof for the invariance of BE-Tree that guarantees safe partitioning and clustering.

**Theorem 3.** The c-node clustering, i.e., bucket splitting, is safe (it avoids the cascading split problem) only if the c-node is associated with an open bucket.

**Proof.** The proof follows simply from the fact that the c-node of an open bucket is not partitioned yet. Hence, the c-node clustering will not result in the cascade splitting problem and the clustering is safe.

**Theorem 4.** The c-node partitioning is safe (it avoids the cascading split problem) only if it is associated with a bucket that is either a discrete bucket or non-leaf bucket.

**Proof.** Based on Lemma 2, a discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting. The safe partition proof follows from the fact that there is no benefit or need to further split a bucket that is the home to all of its expressions. Therefore, the cascading split problem is always avoided. Hence, c-node partitioning of a discrete bucket or non-leaf bucket is always safe.

**Invariance.** Every expression always resides in the smallest bucket that encloses it, and the c-node of a non-discrete leaf bucket is never partitioned.

**Proof.** It follows from Lemma 2 that a discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting, i.e., the smallest bucket that can enclose the expressions. We also need to show this for expressions that reside in a non-discrete leaf bucket, which trivially follows from the definition of a leaf bucket. A leaf bucket is at the lowest level of the clustering directory having no children. Therefore, no other smaller bucket exists in the directory to enclose the non-discrete leaf bucket’s expressions. Hence, every expression always resides in the smallest bucket that encloses it.

It follows from Lemma 1 that a non-discrete leaf bucket is always open. Hence, the c-node of a non-discrete leaf bucket is never partitioned.

**Theorem 5.** BE-Tree’s two-phase space-cutting (space partitioning and clustering) is always safe and always satisfies the BE-Tree invariance.

**Proof.** It follows from the invariance that every expression always resides in the smallest bucket that encloses it. Thus, a discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting. Therefore, partitioning a discrete and non-leaf bucket is always safe according to Theorem 4. In addition, the invariance also enforces that a c-node of a non-discrete leaf bucket is never partitioned because it is not necessarily the home bucket for all of its expressions.

It follows from Theorem 3 that the space clustering is trivially always safe if it applies to only open buckets.

### 3.6.2 BE-Tree Space Complexity

The height of BE-Tree is bounded by $O(k \log N)$, where $k$ is the maximum number of predicates per expression and $N$ is the domain cardinality. Thus, the height of BE-Tree, unlike for other tree-based matching structures such as Gryphon [12], does not grow in the number of subscriptions avoiding memory- and performance-detrimental tree degeneration.
Theorem 6. The height of BE-Tree is bounded by $O(k \log N)$.

Proof. The number of $p$-nodes along each path to an $l$-node is bounded by the number of predicates in each expression. Hence, if there are at most $k$ predicates in each expression, then there are at most $k$ $p$-nodes along each path. Furthermore, the clustering directory is bounded by $O(\log N)$, where $N$ is the cardinality of the domain. This bound is possible due to the discrete and the finite domain. Consequently, the height of BE-Tree is bounded by $O(k \log N)$, where $k$ is the maximum number of predicates per expression.

3.6.3 BE-Tree Operations Time Complexity

The BE-Tree matching follows a typical tree traversal operation in which starting from the root multiple paths of a tree may be traversed until all relevant $l$-nodes are reached; the matching pseudocode is given in Section 3.4.1. In contrast, for insertion, the tree traversal follows exactly one path, which is explained in-depth in Section 3.4.2. The deletion operation is conceptually similar to the matching operation. Finally, the update operation is a composite operation consisting of a deletion followed by an insertion.

Theorem 7. The cost of BE-Tree insertion is bounded by $O(k \log N)$, which is a single-path traversal operation.

Proof. The BE-Tree insertion algorithm chooses the node with the highest score at every level of the tree, which results in a single-path tree traversal. Since the height of the tree is bounded by $O(k \log N)$, then it follows that the BE-Tree insertion is also bounded by $O(k \log N)$.

Theorem 8. The BE-Tree’s matching (i.e., searching) algorithm is a multi-path traversal operation.

Proof. Similar to most multi-dimensional algorithms (such as R-Tree), BE-Tree must follow all relevant paths when matching an event (i.e., searching for all matching subscriptions). Thus, in the worst case, the entire tree must be traversed.

Theorem 9. BE-Tree’s deletion algorithm is a multi-path traversal operation.

Proof. Similar to most multi-dimensional algorithms (such as R-Tree), BE-Tree must follow all relevant paths when searching for a subscription to be deleted. Thus, in the worst case, the entire tree must be traversed.

Theorem 10. The deletion cost of the augmented BE-Tree is bounded by $O(k \log N)$, which is a single-path traversal operation.

Proof. Each subscription is associated with exactly one leaf node. Thus, if the subscription identifier and the leaf node holding it, i.e., the pair (subscription_id, $l$-node), are maintained in a hashtable (having $O(1)$ access time), then the deletion operation can efficiently be implemented using exactly a single-path traversal. As a result, once the subscription’s leaf node is identified using the hashtable and the subscription is removed, then, this information must be propagated upward to the root. This backtracking requires following a single path, and given that the longest path is bounded by $O(k \log N)$, the deletion cost of the augmented BE-Tree is bounded by $O(k \log N)$.

Theorem 11. The cost of the BE-Tree update algorithm is bounded by $O(k \log N)$, which is a single-path traversal operation.
Chapter 3. Query Indexing (Main Memory)

Figure 3.5: BE-Tree Lazy Predicate Evaluation Technique

Proof. The update algorithm is implemented as a deletion operation, i.e., a single-path traversal operation bounded by $O(k \log N)$, followed by an insertion operation, i.e., a single-path traversal operation bounded by $O(k \log N)$; hence, it follows that BE-Tree update cost is also bounded by $O(k \log N)$.  

3.7 BE-Tree Optimizations

In this section, we investigate BE-Tree’s execution model, in order to identify key opportunities to further accelerate the matching computation. First, we present a lazy and a bitmap-based Boolean predicate evaluation. The main focus of the lazy evaluation technique is to ensure exactly-once evaluation of every distinct predicate. On the other hand, the bitmap technique, at the high-level, exploits predicate inter-relationships (i.e., predicate covering) and guarantees exactly-once evaluation of distinct predicates, and at the low-level, it minimizes storage through an efficient bitmap representation and speeds up the computation using low-level bitwise operations and preserves cache locality. Second, we introduce a Bloom filtering strategy to minimize false candidate evaluation at BE-Tree’s leaf level.

3.7.1 Lazy and Bitmap-based Predicate Evaluations

One of the main goals of BE-Tree, in addition to search space pruning, is to minimize the true candidate computations, i.e., the evaluation (and the encounter) of common predicates exactly once. BE-Tree’s structure and cost-function are designed to attain this objective. We generalize the scope of this objective by also ensuring that each distinct predicate is always evaluated exactly once; however, we generalized this exactly-once objective from a different angle. Conceptually in this new paradigm, as we traverse BE-Tree for a given event, we also maintain an efficient structure (with respect to both time and space) to store the predicate evaluation result (True or False) of each distinct predicate in our subscription workload. This structure is represented as a bit-array, in which each bit indicates whether or not a distinct predicate has been evaluated to True (or False). Exploiting a bit-array not only provides fast
read/write access to predicate evaluation results, but also its compact representation, as was observed in most workloads in our experiments, can be pinned entirely in modern processor caches, which significantly reduces the number of cache-misses.

To this end, we propose two different techniques for evaluating predicates using a predicate bit-array. Our first technique, referred to as lazy predicate evaluation, takes a passive approach such that before evaluating a predicate, first, it determines whether or not the predicate has already been evaluated for the current event; if not, then the predicate is evaluated, and the corresponding bit in the predicate bit-array structure is flipped to reflect the predicate evaluation result.

To be precise, we actually maintain two bit-arrays, namely, Pass Bit-array and Fail Bit-array, to maintain predicates that are evaluated to either True or False, respectively. At the outset of matching a new event, both bit-arrays are initialized to zero.\(^2\)

Before evaluating a predicate \(P\), we first check if the bit corresponding to \(P\) is 0 in both Pass/Fail Bit-arrays, i.e., the predicate \(P\) has not been evaluated previously. If the corresponding bit in Pass Bit-array is 1, then it implies that the predicate was evaluated to True for the current event; similarly, if the bit in the Fail Bit-array is 1, then we can infer that the predicate \(P\) is False. Finally, if the predicate \(P\) has not been evaluated previously, then \(P\) is evaluated, and the result is reflected either in Pass or Fail bit-array accordingly. This procedure yields the following invariance regarding Pass/Fail bit-arrays:

**Invariance 1.** For each distinct predicate \(P\), either both bits in Pass/False Bit-arrays are 0 or exactly one bit is set to 1.

The lazy predicate evaluation technique is depicted in Figure 3.5, in which each subscription’s predicate in BE-Tree’s leaf nodes is associated with the predicate bit-arrays. In short, not only the lazy predicate evaluation achieves execution of every predicate exactly once and improves cache-locality but also supports subscription insertions, in which the new subscriptions are permitted to have distinct predicates not seen previously. The new distinct predicates are appended to the end of the bit-arrays. In addition, subscription deletions, which could potentially trigger removal of distinct predicates can be achieved by maintaining an additional supporting structure, Predicate Count Array, such that for each distinct predicate \(P\), it stores the number of subscriptions that contain predicate \(P\). Once a predicate becomes an orphan, then the predicate is marked as deleted (as shown in Figure 3.5). The deleted space is either used to accommodate new distinct predicates or reclaimed periodically through a standard defragmentation (or compacting) procedure. Notably, defragmentation can be done concurrently with event matching, during which the matching solely relies on BE-Tree and does not leverage the lazy predicate evaluation.

Our second predicate evaluation technique, referred to as bitmap-based predicate evaluation, pushes the limit of lazy predicate evaluation by also incorporating the predicate inter-relationships (e.g., predicate covering) through a novel precomputation and storing of predicate coverings, which is achieved partly due to the exploitation of the discrete and finite domain properties. We propose a bitmap structure over the set of all distinct predicates such that for any given attribute-value pair, essentially an equality predicate \(P_{\text{attribute},=\text{value}}\), we precompute and store (designed for an efficient read access in mind) in our bitmap index all distinct predicates that are relevant for \(P\).\(^3\) Therefore, instead of individ-

\(^2\)It is important to note that bit-array initialization can utilize the information from the last seen event in order to selectively reset only parts of the bit-arrays.

\(^3\)Without loss of generality, we focus our discussion on events consisting of only equality predicates, which can easily be extended to support events with range predicates as required in our extended matching semantics by enumerating over the domain of range predicates.
ually evaluating every relevant distinct predicate for a given event’s equality predicate, we precompute the evaluation results of every distinct predicate that is affected by any given event’s predicate. The set of affected predicates by the equality predicate \( P \), denoted by \( \Psi_P \), is defined as follows

\[
\Psi_P = \{ P_i | \forall P_{attr, opt, val} \in \Psi, P_{attr} = P_i^{attr}, \exists x \in \text{Dom}(P_{attr}) \},
\]

where \( \Psi \) is the set of all distinct predicates.

Consequently, the set of all distinct predicates that are not affected by \( P \) are given by \( \overline{\Psi_P} = \Psi - \Psi_P \). The bitmap is constructed by determining the sets \( \Psi_P \) and \( \overline{\Psi_P} \) for each \( P \), i.e., \( P \) is formed by enumerating over the discrete values of each attribute (a dimension in the space) in order to construct an equality predicate \( P \). Next, we evaluate each predicate in \( \Psi_P \) for a given \( P \) and store the results in the bitmap. Also the set \( \overline{\Psi_P} \) is automatically filled with 0 because none of the predicates in the set \( \Psi_P \) are affected by \( P \). The overall structure of the bitmap and its organization of \( \Psi_P \) and \( \overline{\Psi_P} \) are illustrated in Figure 3.6.

The most striking feature of our proposed bitmap is its highly sparse matrix structure because the set of bits represented by \( \overline{\Psi_P} \) are all zero, and given the high-dimensionality of our problem space, we have \( |\Psi_P| \ll |\overline{\Psi_P}| \). Thus, if the corresponding bits are re-ordered by clustering \( \Psi_P \) and \( \overline{\Psi_P} \) (as shown in Figure 3.6), we can achieve an effective space-reduction in our bitmap. This bitmap space saving ratio \( |\Psi_P| / |\overline{\Psi_P}| \), through reordering of the bits in the bitmap, is directly proportional to the number of attributes \( n \) if the predicates are sorted based on their attributes and given by

\[
n \propto \frac{|\Psi_P|}{|\overline{\Psi_P}|}.
\]

This ratio is further influenced by the distribution of predicates over each attribute. For instance, for certain domain values \( \text{value} \) over attribute \( \text{attr} \), there may be no predicate such that the \( \text{value} \) falls in any predicate’s range of values; thereby, implying that the set \( \Psi^{\text{attr}=\text{value}} \) consists of only zero bits. Such a
distribution of predicates results in further space reduction because neither $\Psi_{P_{\text{attr}}}^{=} \cup \Psi_{P_{\text{attr}}}^{\neq}$ need to be stored explicitly in the bitmap. This reduction in space also improves, as a byproduct, the bitwise operations during the matching process. In addition, there are potential research opportunities to develop a more effective bit reordering techniques (i.e., a predicate topological sort order) to further improve the space saving ratio. In particular, the minimum number of predicates that must be maintained for each $P$ (a lower-bound on size of set $\Psi^P$) are those distinct predicates that are satisfied by $P$. This minimum set is defined as

$$\Psi_{\text{min}}^P = \{ P_i | \forall P_{\text{attr}}^{\text{opt}, \text{val}}(x) \in \Psi, P_{\text{attr}} = P_{\text{attr}}^i, \exists x \in \text{Dom}(P_{\text{attr}}), P(x) \land P_i(x) \},$$  

(3.18)

During the BE-Tree matching process, upon arrival of a new event $e$, the precomputed bitmap index is utilized to efficiently compute all distinct predicates that are satisfied by the incoming event. This is carried out by a bit-wise OR-operation of relevant rows in our bitmap index in order to fully construct Result Bit-array: a bit-array in which each bit corresponds to a distinct predicate, where a bit with value 1 signifies that the corresponding predicate is True; otherwise False. The Result Bit-array is constructed as follows:

$$\text{Result Bit-array} = \bigcup_{P_i \in e} \{ \Psi_{P_i}, \overline{\Psi_{P_i}} \},$$  

(3.19)

where no actual operation is required to account for $\overline{\Psi_{P_i}}$ sets.

The Result Bit-array can entirely be pinned in cached as long as the subscription workload contains only in order of few millions of distinct predicates, for which only few mega bytes of cache is required. However, the potential source of cache misses is not limited to Result Bit-array accesses, in fact, another BE-Tree’s internal data structure that generates cache misses (in addition, to general pointer chasing of any tree structure) is the representation of leaf pages content. In BE-Tree with bitmap-based evaluation since each subscription in the leaf page requires only to keep an array of references to Result Bit-array, then it is feasible to store all subscriptions in the leaf page as cache-conscious block of 2-dimensional array of references. Moreover, since in BE-Tree, the number of subscriptions in each leaf page is limited to only tens or hundreds of subscription (cf., Table 3.6), then this 2-dimensional subscription representation could also be fitted in the processor cache; thus, substantially reducing the number of cache misses during subscription evaluations at the leaf level and improving the overall matching time.

In summary, the lazy predicate evaluation technique is most suitable for settings in which many of the expected subscription insertions (or updates) contain unseen distinct predicates while the bitmap-based predicate evaluation technique is ideal for more stable workloads. Therefore, a hybrid mechanism can be adopted in which unseen distinct predicates can be maintained through the lazy predicate evaluation while the stable distinct predicates are maintained through the bitmap technique. Therefore, periodically, the two sets of distinct predicates are merged and the bitmap is reconstructed again. Most importantly, merging the two distinct predicate sets from both the lazy and the bitmap structures can be carried out concurrently as BE-Tree continues to match new incoming events.

\footnote{The processor used in our experiment has two shared 6144KB cache block size.}
3.7.2 Bloom Filtering Optimization

Our final optimization is designed to reduce the number of false candidates at the BE-Tree’s leaf nodes, which is motivated by a simple observation that subscription $s_i$ is a possible candidate if at the very least, the set of attributes on which $s_i$ has defined a predicate explicitly is a subset of attributes appearing in a given event $e$; formally expressed as

$$\forall P_{attr}^{q}, opt, val_{q}^{x} \in s_{i}, \exists P_{attr}^{o}, opt, val_{o}^{x} \in e, P_{attr}^{q} = P_{attr}^{o}. \quad (3.20)$$

This necessary condition $C$ for testing set membership can efficiently be evaluated (approximately) if both the subscription $s_i$ and the event $e$ have a (lossy) compact encoding of all the attributes that appear in $s_i$ and $e$, respectively. To attain this compact encoding, we encode the set of attributes using a Bloom filter representation as follows

$$s_{i}^{\text{bloom}} = \bigvee_{P_{q} \in s_{i}} \text{sdbm\_hash}(P_{q}^{attr}) \mod \text{bloom\_size}, \quad (3.21)$$

where $\lor$ is a bitwise OR-operation, $\text{sdbm\_hash}$ is a hash function, and $\text{bloom\_size}$ is the Bloom filter size. In particular, we utilize the well-known $\text{sdbm\_hash}$ hash function, and we experimented with various choice of $\text{bloom\_size}$ ranging from 16-64 bits in order to investigate the false positive rate of our Bloom filter encoding (cf. Section 3.8). The Bloom filter for the event $e$ is computed in a similar manner.

Therefore, during the matching process, the necessary matching condition is (approximately) satisfied iff the bitwise AND-operation ($\land$) of the subscription’s and the event’s Bloom filters are equal to the subscription’s Bloom filter.

$$C(e, s_i) = \text{True} \iff e^{\text{bloom}} \land s_{i}^{\text{bloom}} = s_{i}^{\text{bloom}} \quad (3.22)$$

3.8 Evaluations

We present a comprehensive evaluation of BE-Tree using both synthetic and real datasets. The experiments were carried on two Quad-core Intel Xeon X5450 processors running at 3.00 GHz machine with 16GB of memory running CentOS 5.5 64bit. All algorithms are implemented in C and compiled using gcc 4.1.2 with optimizations set to O3.

3.8.1 Experiment Overview

We compare BE-Tree with several popular matching algorithms over a variety of controlled experimental conditions: workload distribution, workload size, space dimensionality, average subscription and event size, dimension cardinality, predicate selectivity, dimension selectivity, subscription expressiveness, and event matching probability. In addition, we discuss the effectiveness of sequential BE-Tree in comparison with a state-of-the-art GPU parallel matching algorithm in the context of location-based workloads.

We also ran experiments to determine optimal choices for BE-Tree internal parameters, cf. Section 3.8.5. The values used throughout our experiments are: max$_{cap}$ ranging from 5-160, min$_{size} = 3$, rank$_{window}$ for each node is 10% of expressions in the node’s subtree, and $\theta = 0$. 

---

5This algorithm was created for the $\text{sdbm}$ (a public-domain reimplementation of ndbm) database library. The Bloom filter implementation used is found under http://en.literateprograms.org/Bloom_filter_(C)
Table 3.2: Levels of Expressiveness

<table>
<thead>
<tr>
<th>OpClass</th>
<th>Types of Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>(=)</td>
</tr>
<tr>
<td>Low</td>
<td>(≠, ∈)</td>
</tr>
<tr>
<td>Medium</td>
<td>(&lt;, ≤, =, ≥, ∈, BETWEEN)</td>
</tr>
<tr>
<td>High</td>
<td>(&lt;, ≤, ≠, ≥, ∈, / ∈, BETWEEN)</td>
</tr>
</tbody>
</table>

Table 3.3: Inferred Predicates from \( P_{ij}^{(i,=,v_*)} \)

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i \neq v_1 ) where ( v_1 \neq v_* )</td>
<td></td>
</tr>
<tr>
<td>( i &lt; v_1 ) where ( v_1 \geq v_* )</td>
<td></td>
</tr>
<tr>
<td>( i &gt; v_1 ) where ( v_1 \leq v_* )</td>
<td></td>
</tr>
<tr>
<td>( i \notin {v_1, v_2, \ldots, v_k} \cup {v_*} )</td>
<td></td>
</tr>
<tr>
<td>( i \notin {v_1, v_2, \ldots, v_k} \setminus {v_*} )</td>
<td></td>
</tr>
<tr>
<td>( i ) BETWEEN ( v_1, v_2 ) where ( v_1 \leq v_* ) and ( v_2 \geq v_* )</td>
<td></td>
</tr>
</tbody>
</table>

Once we establish the effectiveness of BE-Tree with respect to state-of-the-art approaches, we shift our focus in the experimental evaluation towards further improving BE-Tree through lazy and bitmap-based predicate evaluations and the Bloom filter optimization.

### 3.8.2 Datasets

**Synthetic Dataset**

One of the key challenges in generating synthetic datasets with high-dimensions is the inability to control the matching probability. With no control mechanism, the matching probability would be virtually zero. To address this concern, we developed a workload generation framework, BEGen. The workloads are generated in two steps: (1) a set of base expressions with only equality predicates are generated, in which a predicate’s attribute is chosen based on either a uniform or a Zipf distribution (which we call the workload distribution); (2) for each base expression \( e_B \), we generate a set of derived expressions, \( e_i \), such that \( \forall P_q(x) \in e_i, \exists P_o(x) \in e_B, P^\text{attr}_o = P^\text{attr}_o, \forall x P_o(x) \rightarrow P_q(x). \) (3.23)

Moreover, each predicate in a base expression is kept with probability \( Pr_{pred} \) in its derived expressions, and each predicate in derived expressions is transformed with probability \( Pr_{trans} \) using one of the inferred predicate rules given in Table 3.3. In our synthetic experiments, base expressions model events, and derived expressions model subscriptions. In addition, the probability \( Pr_{pred} \) and \( Pr_{trans} \) are chosen such that with a high probability, we avoid generating any duplicate subscriptions. Thus, if the average number of predicates per event is \( x \) and the average number of predicates per subscriptions is \( y \), we choose \( x \) and \( y \) such that \( \binom{x}{y} \) is large enough such that duplicate subscriptions are unlikely. Also, the probability \( Pr_{trans} \) further injects variations into the subscription workload. It is controlled by the ratio of equality vs. non-equality predicates within each workload. For example, by tuning the number of generated base expressions, we can control the matching probability for a given subscription workload. For instance, to generate a workload size of 1,000 with matching probability 1%, we generate 100 base expressions and 10 derived expressions for each base expression.

In our evaluation, we assign up to 6 values for \( (\in, \notin) \), and on average, we use a predicate range size of 12% of the domain size for the BETWEEN operator, and we randomly pick a value for the remaining operators. The value of each parameter in our synthetic workload is summarized in Table 3.4, in which
Table 3.4: Experiment Settings for Synthetic Datasets

<table>
<thead>
<tr>
<th>Workload Size</th>
<th>Number of Dimensions</th>
<th>Dimension Cardinality</th>
<th>Predicate Selectivity</th>
<th>Dimension Selectivity</th>
<th>Sub/Event Size</th>
<th>% Equality Predicate</th>
<th>Matching Probability</th>
<th>Extended Matching Semantics</th>
<th>Bloom Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K-1M</td>
<td>50K-100K</td>
<td>100K</td>
<td>100K</td>
<td>100K</td>
<td>100K</td>
<td>100K</td>
<td>100K</td>
<td>100K-1M</td>
<td>100K-1M</td>
</tr>
<tr>
<td>1M</td>
<td>100K</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

Table 3.5: Experiment Settings for Real Datasets

<table>
<thead>
<tr>
<th>DBLP Author</th>
<th>DBLP Title</th>
<th>Matching Probability (Author)</th>
<th>Matching Probability (Title)</th>
<th>Extended Matching Semantics (Author)</th>
<th>Extended Matching Semantics (Title)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>100-760K</td>
<td>50-250K</td>
<td>400K</td>
<td>150</td>
<td>100-760K</td>
</tr>
<tr>
<td>Number Dim</td>
<td>677</td>
<td>677</td>
<td>677</td>
<td>677</td>
<td>677</td>
</tr>
<tr>
<td>Cardinality</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>Avg. Sub Size</td>
<td>8</td>
<td>36</td>
<td>8</td>
<td>36</td>
<td>8</td>
</tr>
<tr>
<td>Avg. Event Size</td>
<td>8</td>
<td>36</td>
<td>16</td>
<td>43</td>
<td>8</td>
</tr>
<tr>
<td>Pred Range Size %</td>
<td>—</td>
<td>—</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>% Equality Pred</td>
<td>—</td>
<td>—</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>Op Class</td>
<td>Min</td>
<td>Min</td>
<td>Lo-Hi</td>
<td>Lo-Hi</td>
<td>Med</td>
</tr>
<tr>
<td>Match Prob. %</td>
<td>—</td>
<td>—</td>
<td>0.01-50</td>
<td>0.01-50</td>
<td>—</td>
</tr>
</tbody>
</table>

each column corresponds to a different workload profile while each row corresponds to the actual value of the workload parameters. Lastly, Table 3.2 captures our four levels of operator expressiveness.

Real Dataset

In the absence of a standard benchmark for evaluating matching algorithm with real data as part of the BEGen framework, we propose a generation of real workloads from extracted public domain data. We focus on the data extracted from the DBLP repository, which is also commonly used as benchmark in assessing algorithms used in the data quality community. In particular, we use the proceeding titles and author names as two sources of data extracted from DBLP.

We, first, use a de-duplication technique to eliminate duplicate entries and to convert the data into a set of \( q \)-grams. This conversion is based on tokenization of a string into a set of \( q \)-grams (sequence of \( q \) consecutive characters). For example, a 3-gram tokenization of “string” is given by \{‘str’, ‘tri’, ‘rin’, ‘ing’\}. Second, we use a transformation to convert each string from a collection of \( q \)-grams into a Boolean expressions. Therefore, we model the collection of \( q \)-grams \{‘str’, ‘tri’, ‘rin’, ‘ing’\} by a set of equality predicates as follows: \[ ‘st’ = ‘r’, ‘tr’ = ‘i’, ‘ri’=’n’, and ‘in’=’g’ \]. This \( q \)-grams-based transformation results in Boolean expressions in a space of 677 dimensions. For the real datasets,

\(^6\)Bibliographic information on major computer science publications.
we also carry out experiments in which we control the degree of matching probability using the profile
generation technique that was used in the synthetic datasets. Table 3.5 summarizes various workloads
generated using the real data sets.

3.8.3 Matching Algorithms

The algorithms in our comparison studies are (1) SCAN (a sequential scan of the subscriptions), (2)
SIFT\(^7\) (the counting algorithm [220] enhanced with the enumeration technique [211] to support range
operators), (3) k-ind (the CNF algorithm implemented over k-index [211]), (4) GR (the Gryphon algo-
rithm [12]), (5) ADGR (our Advanced Gryphon algorithm [12]) which is constructed after applying our
operator transformation given in Table 3.1, (6) P (the Propagation algorithm [74]), (7) APP (the Access
Predicate Pruning algorithm [77] also enhanced with the enumeration technique [211]), (8a) BE (our fully
dynamic version of BE-Tree in which the index is constructed by individually inserting each subscription,
and (8b) BE-B (our batching version of BE-Tree in which all subscriptions are known in advance resulting
in a better initial statistics to guide the space partitioning at the root level). Unlike the construction
of the dynamic BE-Tree, we have constructed k-index, the two versions of Gryphon algorithms (ADGR and
GR), and Propagation using a static workload in which all subscriptions are known in advance.

In addition, we have implemented the specialized GR algorithm that supports only equality predicates
and the generic GR that supports arbitrary predicates [12]; we have included all the Gryphon optimiza-
tions as well [12], i.e., collapsing *-edges (do not care edges) and leveraging predicate covering proposed in [12]
to build the Gryphon data structure. Moreover, after applying our proposed predicate transformation in
Table 3.1, the predicate covering in [12] becomes substantially more effective as demonstrated in our
experimental evaluations.

In our experiments, we distinguish between four levels of predicate expressiveness because not all
algorithms can naturally support our expressive set of operators. In particular, APP [77], SIFT [220],
and k-index [211] naturally support only a weak semantics for operators \(\notin\) and \(\neq\) in which subscription
predicates with inequality on \(\text{attr}_i\) are also matched by an input event that does not define any predicate
on \(\text{attr}_i\); the common alternative semantic is to consider a subscription as matched only if an event
provides a value for all subscription predicates with inequality (strong semantics). To support inequality
operators with the strong semantics, a default value must be added to both subscriptions’ inequality
predicate and the unspecified attributes in the event [211]. This scheme results in an unacceptable
performance as space dimensionality increases for the strong semantics. Thus, we do not consider SIFT
and k-index for the inequality (strong semantics) experiments. Similarly, Propagation does not support
inequality predicates as access predicates and relies on a post processing step to resolve inequalities.
Thus, we do not consider Propagation in the inequality experiments either.

3.8.4 Experiment Organization

In our micro experiments, we study the BE-Tree’s internal parameters and their relation to the overall
performance of BE-Tree. Most importantly, we establish that the maximum l-node capacity is the main
parameter of BE-Tree and provide a systematic guideline on how to adjust it. We then shift gears to
focus on macro experiments in which an extensive evaluation and comparison of BE-Tree with other

---

\(^7\)This counting algorithm is also employed in [142] for a fast GPU implementation.
related approaches are conducted. Finally, we illustrate the effectiveness of BE-Tree's self-adjustment under changing workloads.

In our experiment, the main metric that distinguishes between various matching algorithms is the matching time (i.e., matching response time). In particular, we compare the matching time of BE-Tree with alternative algorithms \( A \) based on the following ratio:

\[
\% = \frac{M^A - M^{\text{BE-Tree}}}{M^A}
\]

where \( M^A \) is the matching time of algorithm \( A \). Therefore, in our discussion, we use this ratio for comparison; for instance, BE-Tree improves over algorithm \( A \) by \( \% \) or BE-Tree reduces the matching time by \( \% \).

### 3.8.5 Micro Experiments

The most important parameter of BE-Tree is the maximum \( l \)-node capacity size, \( \text{max}_{l\text{-node}} \), which triggers our two-phase space-cutting technique. In Figure 3.7 (and Figure 3.8), for various workloads with different matching probability, the effect of varying the \( l \)-node capacity is shown. Although, there is a correlation between the optimal value of maximum \( l \)-node capacity and the degree of the matching probability, the effect is not significant. Thus, we conclude that BE-Tree is not highly sensitive to the maximum \( l \)-node capacity parameter. The results of varying \( l \)-node capacities are summarized in Table 3.6, which we use as a guiding principle throughout our evaluation for choosing the optimal value with respect to
Chapter 3. Query Indexing (Main Memory)

Figure 3.8: Extended Matching Probability with Different $l$-node Capacities

Table 3.6: Most Effective $\text{max}_{cap}$ for Different Matching Probabilities

<table>
<thead>
<tr>
<th>Match Prob</th>
<th>$\text{max}_{cap}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq 1%$</td>
<td>5</td>
</tr>
<tr>
<td>$1% \leq \text{Match Prob} &lt; 10%$</td>
<td>20</td>
</tr>
<tr>
<td>$\geq 10%$</td>
<td>160</td>
</tr>
</tbody>
</table>

the degree of matching probability. Another important factor is that the $l$-node capacity is a tunable parameter which can be dynamically adjusted (increased or decreased) based on the matching feedback to tune future executions of the two-phase space-cutting technique.

Other notable BE-Tree parameters are the base scoring function, the type of clustering directory, and re-insertion. As we described in Section 3.5, for the base scoring function, we used the optimistic popularity measure which resulted in a superior performance in terms of construction time of BE-Tree while reducing the matching computation by further exploiting commonality among subscriptions, the effect of the scoring function popularity ($POP$) and selectivity ($SEL$) is illustrated in Figure 3.9.

Moreover, we considered three strategies for choosing the clustering directory: hybrid clustering directory ($\text{Hy}$), hybrid clustering directory optimized with equality predicate ranking ($\text{Hy-R}$), and generic clustering directory ($\text{Gen}$). Our ranking optimization is an enforcement policy such that all subscriptions’ equality predicates are first consumed by our two-phase space-cutting technique before consuming subscriptions’ non-equality predicates. The rational behind this improvement is two-fold: (1) motivated by our empirical evidence, e.g., $\text{Hy-R}$ improved $\text{Hy}$ and $\text{Gen}$ by up to 52% for the Zipf dataset (Figure 3.10b) and improved $\text{Hy}$ and $\text{Gen}$ by up to 72% for the title dataset (Figure 3.10d) and (2) justified
by utilizing fast hash access specifically designed for equality predicates in our hybrid clustering directory. As a result, for all experiments, we used a hybrid clustering directory with equality ranking except for the experiments with expressive events (events that containing non-equality predicates) in which a generic clustering directory has proven effective. Lastly, we used the adaptive policy and re-insertion policy only for our dynamic experiment in which the re-insertion rate was 5%.

As part of our micro experiments, we also evaluate BE-Tree’s construction time. In particular, the construction time of the dynamic BE-Tree, for the largest workload of up to one million subscriptions, was under 5 seconds in our experiments. BE-Tree’s average construction time and index size, for the representative datasets in our framework, are summarized in Table 3.7.

### Table 3.7: BE-Tree Construction Time & Index Size

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Construction Time (second)</th>
<th>Index Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unif (1M)</td>
<td>2.37</td>
<td>68</td>
</tr>
<tr>
<td>Zipf (1M)</td>
<td>2.03</td>
<td>67</td>
</tr>
<tr>
<td>Author (760K)</td>
<td>3.24</td>
<td>139</td>
</tr>
<tr>
<td>Title (250K)</td>
<td>2.18</td>
<td>69</td>
</tr>
</tbody>
</table>

3.8.6 Macro Experiments

In this section, we compare BE-Tree with several popular matching algorithms over a variety of controlled experimental conditions: workload distribution, workload size, space dimensionality, average subscription and event size, dimension cardinality, predicate selectivity, dimension selectivity, subscription expressiveness, and event matching probability.
Effect of Workload Distribution

The major distinguishing factor among matching algorithms is the workload distribution (the distribution that is used to select a set of attributes for each subscription and each event), which clearly sets apart key vs. non-key based methods. \( k\)-index, APP, and SIFT (non-key-based methods) are highly sensitive to the distribution of the workload whereas BE-Tree and Propagation (key-based methods) are robust with respect to the distribution. The effects of the distribution are shown in Figures 3.11-3.18 in which the graphs on the left column correspond to a uniform distribution while the graphs on the right column correspond to a Zipf distribution. The general trend is that under uniform distribution BE-Tree, ADGR, \( k\)-index, Propagation, APP, GR, and SIFT all outperform SCAN that benefits only from sequential memory access. However, under Zipf distribution both \( k\)-index, APP, and SIFT perform much worse than SCAN. The poor matching time is attributed to few popular attributes that are common among all subscriptions. Therefore, for every event about 80-90% of subscriptions have at least one satisfied predicate which translates into a large number of random memory accesses to increment subscription counters in (APP and SIFT)\(^8\) and scan through \( k\)-index hashtable buckets (referred to as the posting list in [211]). Overall, the BE-Tree matching time is at least four times better than the next best algorithm (\( k\)-index) for uniform distribution and at least two and half times better (Propagation) for Zipf distribution.

\(^8\)However, due to the access pruning method employed in APP, APP outperforms SIFT in all experiments.
Next, we consider the matching time as we increase the number of subscriptions processed. Figure 3.11a,b illustrate the effect on matching time as the number of subscriptions increases in which all algorithms scale linearly with respect to the number of matched subscriptions. In these experiments, BE-Tree exhibits an 80% better matching time as compared to the next best algorithm for the uniform workload and a 63% better matching time for the Zipf workload.

Effect of Dimensionality

Unlike the workload size, the effect of space dimensionality is more subtle; all algorithms with exception of $k$-index, APP, and SIFT are essentially unaffected as the dimensionality varies, see Figure 3.12a,b. The non-key-based algorithms, namely, $k$-index, APP, and SIFT, substantially suffer in lower dimensionality for the uniform workload in which subscriptions tend to share many common predicates, which results in high overlap among subscriptions. Therefore, $k$-index, APP, and SIFT are sensitive to degree of overlap among subscriptions and achieve peak matching time when subscriptions are distributed into a set of disjoint subspaces. For instance, when dimension is set to $d = 50$, BE-Tree improves over $k$-index by 93% and for $d = 1400$, BE-Tree improves over $k$-index by 75% in which $k$-index is the second best algorithm for such high dimensionality. However, for Zipf distribution, see Figure 3.12b, $k$-index, APP, and SIFT matching time does not improve as the dimensionality increases because of the existence of few popular dimensions which results in a large overlap among subscriptions. Therefore, for the Zipf workload,
Effect of Dimension Cardinality

The importance of increasing the dimension cardinality (i.e., the domain size or the number of distinct values in each domain) is twofold: the matching rate and the memory requirement. The matching rate of most algorithms scales gracefully as the dimension cardinality increases, e.g., BE-Tree and Propagation (Figure 3.13). In short, BE-Tree improves over Propagation’s matching time by 66% for the uniform workload and improves over Propagation’s matching time by 59% for the Zipf workload, Figure 3.13a,b, respectively.

However, unlike in BE-Tree, the memory footprint of $k$-index, APP, and SIFT blows up exponentially as we increase the dimension cardinality, while keeping constant the ratio of predicate range size with respect to cardinality. Both approaches rely on the enumeration technique to resolve range predicates. For example, in order to cope with the operator BETWEEN $[v_1, v_2]$, the enumeration essentially transforms the value of $v_2 - v_1$ from a decimal to a unary representation—an exponential transformation. Therefore, we were unable to run $k$-index, APP, and SIFT on workloads with cardinality of 6K and beyond. For instance, the workload with a 6K cardinality has on average a predicate range size of 150 which in turn replaces a single range predicate with 150 equality predicates. To further analyze the role of predicate range sizes, we devise another experiment that varies the predicate range size while fixing the cardinality.

BE-Tree improves over $k$-index by 97%. 

(a) Unif: Dim Cardinality
(b) Zipf: Dim Cardinality

Figure 3.13: Effects of Dimension Cardinality

(a) Unif: Dim Selectivity
(b) Zipf: Dim Cardinality

Figure 3.14: Effect of Dimension Selectivity
Effect of Dimension Selectivity

A notable workload characteristic is the dimension selectivity (i.e., the selectivity of each domain value) of the space, which could have a direct influence on the the ability of matching algorithm to effectively prune the search space. The result of our dimension selectivity is captured in Figure 3.14. For a uniform workload the robustness of BE-Tree and the two versions of Gryphon is evident because as the dimension selectivity varies only a negligible increase in matching time of at most 1% is observed while for Propagation and k-index, a significant increase in matching time of up to 80% and 27% is observed, respectively.

The importance of dimension selectivity is further magnified for the Zipf workload (Figure 3.14b) in which a few dimensions are dominant. Therefore, for a low selectivity, a substantial overhead incurred due to a larger number of false candidates; for instance, as we decreased selectivity from 0.9 to 0.5, the response time is increased by 411% and 650% for Propagation and k-index, respectively. In general, the low selectivity results in a less effective pruning of the search space, and BE-Tree compensate for the low selectivity side effect by a deeper tree structure that provides a greater opportunity to prune the search space.

Effect of Predicate Selectivity

In fact, the ratio of predicate range size with respect to the dimension cardinality is inversely proportional to the predicate selectivity. The predicate selectivity has a small influence on Propagation, which relies solely on selective equality predicates, while it has a huge influence on k-index, APP, and SIFT, which do not utilize the predicate selectivity information. Therefore, as shown in Figure 3.15, as the ratio of the predicate range size increases (selectivity decreases), the search space pruning mechanism of k-index and SIFT suffer due to the increased number of false candidates.

In general, a low selective predicate causes a less effective pruning of the search space, and BE-Tree compensates for the low selectivity with a deeper tree structure that provides a greater opportunity to prune the search space by using both highly selective predicates (equality) and low selective predicates (range operators). As a result, BE-Tree improves over the next best algorithms by 76% and 43% for the uniform and the Zipf workloads, respectively (cf. Figure 3.15).
Another key workload characteristic is the average number of predicates per subscription and event. We analyzed the effect of subscription and event size with respect to three different workload characteristics: varying both subscription and event size (Figures 3.16a,b), varying subscription size while fixing event size (Figures 3.16c,d), and varying event size while fixing the subscription size (Figures 3.16e,f).

In particular, $k$-index, APP, and SIFT are highly sensitive to the number of predicates: in addition to increasing the overlap among subscriptions, for $k$-index, it also translates into a longer sorting time, and for APP and SIFT, it translates to a larger number of retrieving and scanning hashtable buckets when considering increasing both subscription and event size, Figures 3.16a,b. The Propagation algorithm starts with a higher matching time because subscriptions have fewer predicates and the chances of finding an equality access predicate with high selectivity is lower, as subscriptions are not evenly distributed in space. As a result, the Propagation algorithm reaches its optimal performance when average subscription
size reaches 14, and no noticeable benefit is gained as the subscription size further increases, instead the response time gradually increases due to an increase in computation cost for checking each predicate. In general, BE-Tree gracefully scales as the number of predicates increases because of its multi-layer structure and improves over the next best algorithm by 63% for the uniform and by 65% for the Zipf workload, as illustrated in Figures 3.16a,b.

When we vary the subscription size while fixing the event size (Figures 3.16c,d), the key observation is that the increase in the number of subscription’s predicates has less impact on matching time as opposed to the increase in the number of event’s predicates. The cost associated with increasing the number of subscription’s predicates is evident because a longer time is required to check a larger number of predicates. As shown in figure c-d, varying the number of subscription/event predicates from 5/81 to 5/81 results only in linear increase in matching time. In fact, for $k$-ind, under Zipf distribution, the matching time is even slightly reduced as the number of subscription increases because it provides a better opportunity to prune the search space. Remarkably, $k$-ind reaches its worse performance when the number of event’s predicates is increased, when moving from Figures 3.16a to 3.16c, BE-Tree improves $k$-ind by 72% and BE-Tree improves $k$-ind by 98%, respectively. This observation is more striking in Figure 3.16e,f, which is discussed next.

In Figure 3.16e,f, when increasing the number of event predicates from 13 to 81 $k$-ind matching time is increased by 42.6 times while BE-Tree, SCAN, P, and APP increased by less than 3 times. In conclusion, BE-Tree was dominant throughout all experiments as we varied subscription and event sizes.

**Effect of Percentage of Equality Predicates**

In this experiment, we study the effects of ratio of equality vs. non-equality predicates for each subscription. The general trend is that the matching time for all algorithms improve as the percentage of subscription equality predicates increases because the overlap among subscriptions is reduced, see Figure 3.17a,b. Most notably, when subscriptions consist only of equality predicates, the specialized GR (or our proposed ADGR) (for equality predicates) results in a substantial performance gain, being the best algorithm after BE-Tree, compared with the generic GR. The Propagation algorithm also improves significantly when subscriptions are restricted to only equality predicates because there is a better chance to find more effective access predicates. However, among all algorithms, GR and Propagation (but not ADGR) are the most sensitive algorithm with respect to the percentage of equality predicates; as the percentage of equality predicate decreases, their performance substantially deteriorates. For instance, for
the uniform workload, Figure 3.17a, BE-Tree improves over GR matching time by 88% when subscriptions only have few equality predicates and by 20% when subscriptions restricted to only equality predicates.

Effect of Percentage of Matching Probability

As the matching probability increases, the number of candidate subscriptions and the event matching time also increases. Therefore, we studied the effects of varying matching probability under both uniform and Zipf workload based on different levels of predicate expressiveness. Under a uniform workload with low and medium expressiveness, while keeping the matching probability below 1%, \( k \)-index outperforms Propagation, APP, and SIFT while BE-Tree improves over \( k \)-index by 97%. However, as the matching probability goes beyond 3%, the Propagation algorithm begins to outperform \( k \)-index, APP, and SIFT while BE-Tree improves over ADGR and Propagation by up to 33%, even as the matching probability reaches 9%, see Figure 3.18a,c; as the matching probability goes beyond 35%, the success of BE-Tree continues
as **BE-Tree** becomes marginally the better algorithm followed by **Propagation** and **SCAN**, Figure 3.20a,b.

In general, an increase in matching probability results in an increase in the number of candidate matches; therefore, **APP** and **SIFT** is forced to scan large hashtable buckets, using random access, to increment subscription counters for each of the satisfied predicates. Similarly, **k-index** is forced to scan large buckets (i.e., posting lists) with a reduced chance of pruning and an increased application of sorting to advance through each bucket. For the Zipf distribution, **Propagation** remains the next best algorithm after **BE-Tree**, see Figure 3.18b,d. Furthermore, in the experiments where the highest level of expressiveness was used, **BE-Tree** dominates both versions of **Gryphon** algorithms and **SCAN** by orders of magnitude, see Figure 3.18e,f.
Real Datasets Experiments

In the evaluation over real datasets, extracted from DBLP, we first considered varying the workload size without controlling the matching probability and the predicate expressiveness. Therefore, subscription and event workloads were constructed by a direct translation from string data to $q$-gram and ultimately into a conjunction of equality predicates. Second, we considered the effects of changing the matching probability followed by event expressiveness.

In our synthetic experiments in which the percentage of equality predicates is varied, BE-Tree and the two versions of Gryphon algorithms are the top performing algorithms followed by Propagation and $k$-index. Similar trends were also observed for real datasets. In particular, for the author dataset, Figure 3.21a, with an average of 8 predicates per subscriptions, BE-Tree improves over GR by 37% while more substantially improving over Propagation by over 98%. For the title dataset with much larger number of predicates per subscriptions, that is at around 35 predicates per subscriptions, the gap between BE-Tree and the other algorithms further widens. This is due to BE-Tree’s scoring that exploits interrelationships within dimensions and the multi-layer structure of BE-Tree that effectively utilizes most of the subscription predicates to reduce the search space. Therefore, as demonstrated in Figure 3.21b, BE-Tree improves over GR by 51% and significantly improves over Propagation by more than 99%, up to three orders of magnitude. Furthermore, we have conducted the matching probability experiments with varying degree of expressiveness which produced similar results as in our synthetic dataset, which is shown in Figure 3.19. Moreover, for the author dataset, as the matching probability goes beyond 50%, SCAN becomes marginally the better algorithm followed by BE-Tree and Propagation Figure 3.20c.
However, for the title dataset, even as the matching probability reaches 50%, **BE-Tree** outperforms both **SCAN** and **Propagation** by nearly 34%, Figure 3.20d. This performance gain is due to a larger average expression size (a larger number of predicates in each expression) in the title dataset, which increases **BE-Tree**’s benefits by pruning the search space.

**Effect of Event Expressiveness**

One of the distinct features of our matching semantics and **BE-Tree** is to support an expressive event language because our proposed semantics does not differentiate between event and subscription and models both as Boolean expressions. Hitherto, we focus on evaluating stabbing subscription model in which event expressions were limited to only equality predicate, a model which is adopted by most prior work. Therefore, next, we shift the focus to stabbing subscription model in which the event is no longer limited to only equality predicate.

Notably, the only other relevant matching algorithm that naturally supports expressive event expression is **SCAN**. In addition, the **Propagation** can internally be augmented in order to support a more expressive event expression, but such augmentation is non-trivial especially for non-key based counting approaches such as **Gryphon**, **k-index**, **APP**, and **SIFT**. Our proposed augmentation for **Propagation** is as follows: for each event, non-equality predicates are treated as a set of equality predicates by enumerating over the predicate permitted range of values. For instance, during the runtime the predicate $[\text{attr}_i \text{ BETWEEN} [a,b]]$ is replace by

$$[\text{attr}_i = a, \text{attr}_i = a + 1, \cdots, \text{attr}_i = b].$$ (3.25)

Therefore, we evaluated **BE-Tree**, **Propagation**, **SCAN** under this new paradigm while varying the workload size and the distributions using both synthetic and real datasets. The experimental results are demonstrated in Figure 3.22. As expected, **BE-Tree** continues to outperform both **Propagation** and **SCAN**, and the gap between **BE-Tree** and **Propagation** is further widen because of the natural support of **BE-Tree** for expressive event expressions through its two-phase space-cutting technique, and, in particular, its *generic clustering directory* structure. For the synthetic datasets, under uniform workload, **BE-Tree** improves over **Propagation** by up to 98% (Figure 3.22a) and under Zipf workload **BE-Tree** remains superior and improves over **Propagation** by up to 56% (Figure 3.22b). The smaller gap under Zipf workload is due to the nature of the workload in which not only few dimensions are dominant, but also due to allowing
non-equality predicate in event expression, for those dominant dimensions, an event expression spans a large range of domain values. Consequently, there is a higher degree of matching probability, which is inherently harder to control, under the Zipf workload compared to the uniform workload. Similarly, for the real datasets, BE-Tree also significantly outperforms Propagation by 89% (Figure 3.22c) for the author dataset and by 96% (Figure 3.22d) for the title dataset.

Adapting to Subscription/Event Changes

In our adaptive experiments, we studied the self-adjusting mechanism and the maintenance cost (e.g., insertion, deletion, update) of BE-Tree. The experiment setup is as follow. First, we fixed the event workload in which half of the events are generated using a uniform while the other half using a Zipf distribution. Second, we generated uniform and Zipf subscription workloads with 0.1% matching probability. In each experiment, we start by individually inserting each subscription from workload $X$ into BE-Tree, then we individually remove each subscription from BE-Tree and individually insert subscriptions from the workload $Y$ until BE-Tree contains only subscriptions from workload $Y$, then we reverse this process until we gradually switch back to the original workload $X$. After a fixed number of deletions and insertions, we run our event workload, and record the matching time. The objective is to illustrate that BE-Tree adapts to workload changes and the performance of BE-Tree does not significantly deteriorate even in extreme situations in which the distribution rapidly and dramatically changes. In the first experiment, while transitioning from uniform to Zipf and back again, the matching time at the end of the transition approaches the original performance, Figure 3.23a. A similar adaptation was observed
when we transitioned from Zipf to uniform and back again, Figure 3.23b. Another important observation from these experiments is that matching cost, as expected, is the dominant compared to insertion and deletion cost. Also in BE-Tree, the update operation is simply implemented as a deletion followed by an insertion.

The above experiments mostly focused on adaptation to the drastic changes of the subscription workload. Next we consider the BE-Tree’s adaptation with respect to event workload by utilizing BE-Tree’s cost function. In these experiments, we consider two types of event workload: (1) an event stream that results in generating up to 20% false candidate, by altering and adding holes in subscriptions’ covered predicates, (Figure 3.24a) and (2) an event stream that generates up to 80% false candidate (Figure 3.24b). For both experiments, an identical subscription workload is used, and subscriptions are inserted incrementally in 100k batches followed by re-executing the entire event stream over it. We perform the event matching using two BE-Tree variations: BE-ADT in which $\beta$ in our cost function is tuned dynamically as described in Section 3.5, and BE in which $\beta$ is set to 0.5 and is fixed. We can observe that BE-ADT can detect the high rate of false candidate generated by covered predicates; therefore, when BE-Tree is undergoing the two-phase space partitioning for the newly inserted subscriptions, it attempts to utilize subsumed predicate and avoid using covered predicates. Consequently, as more subscriptions are inserted the gap between BE-ADT and BE widens, and a matching time reduction of up to 10% is obtained (Figure 3.24b).
Table 3.8: BE-Tree/Bitmap Construction Time (second) & Memory Usage (MB)

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Number of Distinct Pred.</th>
<th>Construction Time</th>
<th>Index Size</th>
<th>Construction Time</th>
<th>Bitmap Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unif (1M)</td>
<td>519607</td>
<td>26.67</td>
<td>60</td>
<td>23.01</td>
<td>3.2</td>
</tr>
<tr>
<td>Zipf (1M)</td>
<td>266132</td>
<td>3.75</td>
<td>57</td>
<td>97.92</td>
<td>1.7</td>
</tr>
<tr>
<td>Author (760K)</td>
<td>10810</td>
<td>4.47</td>
<td>98</td>
<td>0.97</td>
<td>0.2</td>
</tr>
<tr>
<td>Title (250K)</td>
<td>11566</td>
<td>2.29</td>
<td>37</td>
<td>1.23</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Figure 3.25: Effects of Dimensionality/Dimension Cardinality on Lazy/Bitmap Optimizations

3.8.7 BE-Tree Optimization Evaluation

In this section, we study the effectiveness of key BE-Tree optimizations including bitmap evaluation and Bloom filter pruning.

Lazy and Bitmap-based Predicate Optimizations

We begin by investigating BE-Tree (batch version) and bitmap construction time and memory usage that are summarized in Table 3.8. First, we observe that the construction time of BE-Tree with batch processing is longer compared to dynamic BE-Tree (which was shown in Table 3.7) because all subscriptions are known in advance; thus, statistics gathering and computations take longer, which in turn results in improved matching time. Second, for the Zipf distribution, subscriptions are densely distributed, i.e., many common and overlapping predicates, which result in few dense regions of space associated to a large number predicates. On the one hand, these densely distributed regions result in a higher bitmap construction time while, on the other hand, the fewer number of distinct predicates that are densely distributed across the attribute space result in a more effective space saving ratio and ultimately reduced memory requirements of the bitmap structure. These trends are also captured in Table 3.8.

Next, we demonstrate experimentally the robustness of our proposed lazy and bitmap-based predicate
evaluation techniques, with respect to key workload parameters that may affect its outcome, including workload distribution (Figures 3.25-3.26), space dimensionality (Figure 3.25a,b), dimension cardinality (Figure 3.25c,d), event matching probability (Figure 3.26), and average subscription and event size (Figure 3.27).

In order to investigate the scalability of our BE-Tree’s proposed optimizations, we experiment with varying the number of dimension and dimension cardinality. As we scale the number of dimension from 50-1400 not only do these optimizations scale well with respect to memory use, but they also improve the matching time by up to 29% and 55% for lazy and bitmap techniques, respectively, (Figure 3.25a,b). Likewise, as we increase the dimension cardinality from 48-150K, our lazy and bitmap techniques continue to outperform BE-Tree by up to 9% and 57%, respectively, (Figure 3.25c,d).

In order to judge the broad applicability of BE-Tree extended with lazy and bitmap optimizations, we study the effect of varying the degree of matching probability for both synthetic and real workloads. These optimizations substantially outperform the BE-Tree structure as we increase the matching probability because a higher matching probability translates into fewer number of distinct predicates, which in turn is beneficial to both techniques. As a result, on average, with the lazy predicate evaluation, we obtain up to 43% improvement while the bitmap approach significantly reduces BE-Tree’s matching time by up to 64%, as shown in Figure 3.26.

A key distinguishing workload parameter that demonstrates the effectiveness of our lazy and bitmap evaluations is the effect of changing subscription and event size because as we increase the number of predicates per subscription, the number of predicate evaluations also increases resulting in a larger saving
from evaluating every distinct predicate exactly once. As a result, for the lazy optimization, on average, we reduce the matching time by up to 24% for both uniform and Zipf distributions while for the bitmap optimization, on average, we achieve up to 75% improvement for both workloads (Figure 3.27).

The final aspect of our bitmap optimization is to demonstrate the effectiveness of our 2-dimensional subscription representation. Therefore, we consider two variations of BE-Tree with bitmap optimization: bitmap, which is the base version, and bitmap-CC, which is the cache-conscious version that incorporates 2-dimensional subscription representation. As discussed previously, with 2-dimensional representation both subscriptions and Result Bit-array could potentially fit in the processor cache entirely and eliminate all cache misses. This reduction in the number of cache misses is clearly evident by significant reduction in matching time by up to 70% as shown in Figure 3.28.

**Bloom Filter Optimization**

We conclude our experimental studies with an analysis of the Bloom filter optimization. In these experiments, as we increase the workload size, we vary the Bloom filter size from 16-64 bits, as shown in Figure 3.29. As expected, the increase in the Bloom filter size decreases the false positive rate of the Bloom filter, which in turn, results in an improved matching time by eliminating false candidate matches. However, as the Bloom filter size is increased, the overhead for checking the necessary condition C through the Bloom filter is also increased. Thus, in our experiment, a 32-bit Bloom filter achieved the right balance between the size and reduction in false positive rate. For instance, when a 32-bit Bloom filter is used, the matching time of BE-Tree with lazy optimization is improved by 70%
and 50% for uniform and Zipf distributions (Figure 3.29a,b) while the matching time of BE-Tree with bitmap optimization is improved by 21% and 16% for uniform and Zipf distributions (Figure 3.29c,d). As expected, the Bloom filter optimization is more effective for lazy optimization, in which the cost of predicate evaluation is higher compared to when the bitmap optimization is used instead; hence, the benefit of false candidate reduction of Bloom filter is more prominent in conjunction with the lazy predicate evaluation. In general, based on our findings, the Bloom filter optimization is best suited when matching probability is low, i.e., event matches are rare, and, more importantly, there are only handful of predicates (or attribute-value pairs) in each event.

3.8.8 Comparison With A GPU-based Algorithm

In this final experiment, two major algorithms were considered. Our PC-based algorithm, namely, BE-Tree, a single-threaded algorithm, and a GPU-based algorithm, for which we have drawn results from the GPU-based algorithm CLCB ran on Nvidia GTX 460 with 1GB of memory using the CUDA Toolkit 4.1 [58].

We experiment with two of our publicly released BE-Tree algorithms. BE-Tree 1.1 (which includes our bitmap optimization), whereas BE-Tree 1.3\(^9\), which further improves low-level implementations and includes both bitmap optimization and 2-dimensional subscription representation. The Bloom filter optimization is not included in BE-Tree 1.3.

We rely on default location-based workloads used in [58], for serving as a common benchmark. The

\(^9\)The BE-Tree 1.1 and 1.3 binaries are available on [http://msrg.org/project/BE-Tree](http://msrg.org/project/BE-Tree)
workload consists of executing 1000 events over 2.5 million subscriptions. Both events and subscriptions consist of 3-5 predicates drawn uniformly from 100 dimensions, where the cardinality of each domain is 65K. The subscriptions and events in the location-based workload have only 2-4 regular predicates plus an additional location-based predicate. Since the location predicates generated in [58] have higher selectivity, the average number of matched subscriptions are increased from 3 to 70 when moving from the workloads without to with location predicates.

The experimental results are summarized in Table 3.9. The CLCB algorithm (ran on GPUs) on average processes each event in 0.306 ms for location-based workloads while BE-Tree 1.3 can sustain an average matching time of 0.045 ms for non-location-based workload and matching time of 0.067 ms for location-based workload (nearly 5x faster than CLCB). The slight increase in BE-Tree matching is due to increased number of matched subscriptions.

Therefore, contrary to the reported results in [58] (based on BE-Tree 1.1), in fact, not only BE-Tree’s matching time does not deteriorate when adding location-based information (because BE-Tree makes no distinction between location and non-location-based predicates and BE-Tree’s underlying novel two-phase space-cutting technique can utilize all types of predicates), but BE-Tree also outperforms the CLCB algorithm ran on GPUs [142, 58]. The incorrect BE-Tree results reported in [58] was due to supplying BE-Tree with a wrong parameter i.e., enabling a simple heuristic optimization, not visible to the authors in [58]; thus, the authors were unaware of this misconfiguration [58]. This simple heuristic optimization stated that for each subscription, first all equality predicates are used in BE-Tree’s two-phase space-cutting technique before considering the non-equality predicates because, in general, the equality predicates tend to have a higher selectivity. Surprisingly, the workload generated in [58] had a peculiar distribution, which acted as an adversary to this simple ad-hoc heuristic that is not at the core of BE-Tree. In addition, the adaptive self-adjusting mechanism of BE-Tree was disabled.

### 3.8.9 Experimental Summary

To summarize our evaluation, let us consider three main workload categories: (1) workloads with uniform distribution and a low-to-high degree of expressiveness, (2) workloads with Zipf distribution and a low-to-high degree of expressiveness, and (3) real-world and synthetic workloads with minimum degree of expressiveness (equality predicates only). From best to worst performing algorithms, in the first category (uniform) we have: BE-Tree, our Advanced Gryphon (ADGR), k-index [211], Propagation [74], APP [77], and SIFT [220]. In the second category (Zipf) we have: BE-Tree, Propagation, ADGR, k-index, APP, and SIFT. Lastly, in the third category we have: BE-Tree, ADGR, Gryphon [12], Propagation, and k-index. The general trends are that non-key based algorithms, i.e., k-index and SIFT, do poorly on workloads that consist of few popular dimensions (i.e., low dimensional space and Zipf distribution) because of the significant increase in the number of false candidates that have to be considered. Also, Gryphon and Propagation are highly sensitive to the degree of expressiveness of subscriptions. Finally, BE-Tree dominated in every category, yet by incorporating our lazy and bitmap-based predicate evaluation optimizations, BE-Tree’s matching time is further reduced by up to 43% and 75%, respectively. We can shape off up to an

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>BE-Tree 1.1</th>
<th>BE-Tree 1.3</th>
<th>CLCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>without location</td>
<td>0.081 ms</td>
<td>0.045 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>with location</td>
<td>0.144 ms</td>
<td>0.067 ms</td>
<td>0.306 ms</td>
</tr>
</tbody>
</table>
additional 70% of the matching time of BE-Tree (which is already extended with our predicate evaluation optimizations) after also applying the Bloom filter optimization when matching probability is low.

3.9 Summary

In this chapter, we presented BE-Tree, a novel index structure to efficiently index and match Boolean Expressions defined over a high-dimensional discrete space. We introduced a novel two-phase space-cutting technique to cope with the curse of dimensionality underlying the subscription and event space, which appears in many application domains. Furthermore, we developed an effective cost model and self-adjustment policies that enable BE-Tree to actively adapt to workload changes. Moreover, we proposed scalable and effective predicate evaluation techniques, i.e., lazy and bitmap, and a Bloom filter optimization, which substantially improve BE-Tree’s matching computation. Finally, through an extensive experimental evaluation, we demonstrated that BE-Tree is a generic index for Boolean expressions that supports variety of workload configurations and handles predicates with expressive set of operators.

Consequently, we presented a wide range of applications, including distributed event-based systems, applications in the co-space, targeted web advertising, and approximate string matching, that can benefit from a general purpose indexing technique for Boolean expressions.
Chapter 4

Parallel Query Indexing (Multi-core)

4.1 Introduction

Efficient event processing (i.e., event matching) is an integral part of a growing number of web and data management technologies ranging from user-centric processing and personalization to real-time data analysis. Among user-centric processing applications, there are computational advertising [211, 82], online job matching sites [140, 211], and location-based services for emerging applications in co-spaces [11, 160]. Common among all of them are patterns and specifications (e.g., advertising campaigns, job profiles, service descriptions) modeled as Boolean expressions, XPath expressions, or SQL queries and incoming user information (e.g., user profiles and preferences) modeled as events using attribute-value pairs or XML documents. In the real-time analysis domain, there are (complex) event processing [12, 74, 176, 82, 178], XML filtering [14, 67], intrusion detection [196], and computational finance [182]. Again, common among these applications are predefined sets of patterns (e.g., queries and attack specifications) modeled as subscriptions and streams of incoming data (e.g., relational tuples, data packets) modeled as events.

Unique to user-centric processing and personalization are strict requirements to determine only the relevant content (e.g., ads) with respect to user demographics and interests [140, 211, 82]. Furthermore, user-centric processing demands scaling to millions of patterns and specifications (e.g., advertising campaigns) and millions of users (i.e., consumers) while meeting processing latency constraints in the subsecond range, to achieve an acceptable service-level agreement. Noteworthy, users often within certain demographic proximity are potentially interested in similar content and demand receiving the same contents (e.g., following popular local news, songs, or products).

We argue that in order to meet these scaling requirements, it is essential to develop efficient parallel event matching algorithms that are compatible with state-of-art high-dimensional indexing structures [176, 179, 178] and are capable of extracting and exploiting similarity among incoming events. In other words, matching algorithms that are able to efficiently identify and aggregate users’ profiles with similar interests and amortize the cost of finding user-relevant content over many users.

Unique to real-time data analysis applications are critical requirements to meet the ever growing demands in processing large volumes of data at predictably high-throughput and low latencies across many application scenarios [196, 144, 103, 63]. The need for increased processing bandwidth is the key ingredient in high-throughput real-time data analysis that enables processing, analyzing, and extracting
relevant information from streams of incoming data. We argue that in order deliver real-time event processing in presence of a continuous proliferation of data and bandwidth growth, it is inevitable to expand the research horizon beyond that of conventional sequential matching algorithms and adopt other key enabling technologies such as multi-core architectures. There is mounting evidence that the number of cores is predicted to grow exponentially in the coming years [117, 166], and that multi-cores are successfully being used to accelerate many data management applications [197, 118]. Therefore, the next generation of real-time event processing must exploit parallel hardware such as multi-core architectures to sustain the expected growth in demands.

These requirements constitute challenges of paramount importance for applications that rely on event processing (with primary focus on patterns defined as Boolean expressions). To address these challenges, first and foremost, we develop a novel parallel event matching algorithm over an exiting state-of-the-art Boolean expression index structure [176, 179, 178]. Our parallel matching algorithm exploits multi-threading and the shared-memory architecture of modern multi-core processors in order to scale to millions of expressions and to meet the high-throughput demands. Such parallel matching algorithms have received little attention by prior Boolean expression matching approaches (e.g., [12, 74, 140, 211, 82, 176, 178]). Second, we introduce a stream-aware adaptive parallel algorithm that can identify and utilize the overlap (or the similarity) within the event stream. Similarly, online event stream re-ordering techniques have also been largely ignored by the plethora of prior-art on event matching algorithms (e.g., [12, 74, 140, 211, 82, 176, 178]).

In short, the contributions of this chapter are four-fold: (i) We develop a novel Parallel Compressed event Matching (PCM) algorithm over a bitmap-based event encoding detailed in Section 4.3. (ii) We develop an efficient Online Stream Re-ordering (OSR) technique presented in Section 4.4. (iii) We develop an Adaptive Parallel Compressed event Matching (A-PCM) algorithm that adaptively chooses between matching over compressed or uncompressed events depending on the stream similarity discussed in Section 4.4. (iv) Finally, we conduct a comprehensive experimental evaluation that establishes the effectiveness of our approach in Section 4.5.

4.2 BE-Tree Bitmap Encoding

In this section, we focus on BE-Tree’s execution model that exploits opportunities to further accelerate the matching computation that relies on a bitmap-based event encoding [179]. The main advantage of the bitmap-based technique is to ensure exactly-once evaluation of every distinct predicate. At a high-level, this technique exploits predicate inter-relationships (i.e., predicate covering) and guarantees exactly-once evaluation of distinct predicates. At a low-level, it minimizes the memory footprint through an efficient bitmap representation that also speeds up the computation using low-level bitwise operations to preserve cache locality.

One of the BE-Tree properties, in addition to search space pruning, is minimizing the true candidate computations, i.e., the evaluation (and the encounter) of common predicates exactly once. BE-Tree’s structure and cost-function are designed to attain this objective. The scope of this objective is generalized by also ensuring that each distinct predicate is always evaluated exactly once. However, this exactly-once objective is generalized from a different angle. Conceptually, in this paradigm, as BE-Tree is traversed for a given event, an efficient structure is maintained (with respect to both time and space) to store the evaluation result (True or False) of each distinct predicate in the subscription workload. This structure
is represented as a bit-array, in which each bit indicates whether or not a distinct predicate has been evaluated to \textbf{True} (or \textbf{False}). Exploiting a bit-array not only provides fast read/write access to predicate evaluation results, but also its compact encoding could potentially be resident entirely in the cache of a modern processor, which significantly reduces the number of cache-misses.

Therefore, the \textit{bitmap-based event encoding}, pushes the limit of \textit{BE-Tree}'s standard predicate evaluation by also enabling exactly-once evaluation and incorporating the predicate inter-relationships (e.g., predicate covering) through a novel precomputation and storing of predicate coverings, which is achieved partly due to the exploitation of the discrete and finite domain properties. The bitmap structure is over the set of all distinct predicates such that for any given attribute-value pair, i.e., the equality predicate \( P_{\text{attr}, \text{opt}, \text{val}} \), all distinct predicates that are relevant for \( P \) are precomputed and stored in the bitmap. Therefore, instead of individually evaluating every relevant distinct predicate for a given event’s predicate, the evaluation results are precomputed for every possible distinct predicate that is affected by any given event’s predicate. The set of affected predicates by the equality predicate \( P \), denoted by \( \Psi^P \), is defined as follows

\[
\Psi^P = \left\{ P_i \mid \forall P_i^{\text{attr}, \text{opt}, \text{val}}(x) \in \Psi, P_i^{\text{attr}} = P_{\text{attr}}, \exists x \in \text{Dom}(P_{\text{attr}}) \right\},
\]

where \( \Psi \) is the set of all distinct predicates.

Consequently, the set of all distinct predicates that are not affected by \( P \) are given by \( \overline{\Psi^P} = \Psi - \Psi^P \). The bitmap is constructed by determining the sets \( \Psi^P \) and \( \overline{\Psi^P} \) for each \( P \), i.e., \( P \) is formed by enumerating over the discrete values of each attribute (a dimension in the space) in order to construct an equality predicate \( P \). Next, each predicate is evaluated in \( \Psi^P \) for a given \( P \) and stored as the result in the bitmap. Also, the set \( \overline{\Psi^P} \) is automatically filled with zeros because none of the predicates in the set \( \overline{\Psi^P} \) are affected by \( P \). The overall structure of the bitmap and its organization of \( \Psi^P \) and \( \overline{\Psi^P} \) are illustrated in Figure 4.1.
The most striking feature of the bitmap structure is its highly sparse matrix structure because the set of bits represented by $\Psi^P$ are all zero, and given the high-dimensionality of our problem space, the bitmap is space-efficient because $|\Psi^P| \ll |\Psi^P|$. In particular, if the corresponding bits are re-ordered by clustering $\Psi^P$ and $\Psi^P$ (as shown in Figure 4.1), an effective space-reduction can be achieved for the bitmap structure. The bitmap space saving ratio $\frac{|\Psi^P|}{|\Psi^P|}$, through reordering of the bits in the bitmap, is directly proportional to the number of attributes $l$ if the predicates are sorted based on their attributes and given by

$$l \propto \frac{|\Psi^P|}{|\Psi^P|}.$$  

This ratio is further influenced by the distribution of predicates over each attribute. For instance, for certain domain values value over attribute $attr$, there may be no predicate such that the value falls in any predicate’s range of values; thereby, implying that the set $\Psi^{attr,=,v}$ consists of only zero bits. Such a distribution of predicates results in further space reduction because neither $\Psi^{attr,=,v}$ nor $\Psi^{attr,=,v}$ need to be stored explicitly in the bitmap. This reduction in space also improves, as a byproduct, the bitwise operations during the matching process. In addition, there are potential research opportunities to develop more effective bit reordering techniques (i.e., a predicate topological sort order) to further improve the space saving ratio. In general, the minimum number of predicates that must be maintained for each $P$ (a lower-bound on size of the set $\Psi^P$) are those distinct predicates that are satisfied by $P$. This minimum set is defined as follows:

$$\Psi^P_{\min} = \{P_i | \forall P^i_{attr,opt,val}(x) \in \Psi, \ P^i_{attr} = P^i_{attr}, \exists x \in \text{Dom}(P^i_{attr}), \ P(x) \land P^i_i(x)\},$$

During the BE-Tree matching process, upon arrival of a new event $e$, the precomputed bitmap index is utilized to efficiently compute all distinct predicates that are satisfied by the incoming event. This is carried out by a bit-wise OR-operation of relevant rows in the bitmap index in order to fully construct the Result Bit-array: A bit-array in which each bit corresponds to a distinct predicate, where a bit with value 1 signifies that the corresponding predicate is True, otherwise False. The Result Bit-array is constructed as follows:

$$\text{Result Bit-array} = \bigcup_{P_i \in e} \{\Psi^P_i, \overline{\Psi^P_i}\},$$

where no actual operation is required to account for $\Psi^P_i$ sets.

The Result Bit-array, namely, the bitmap-based event encoding, can entirely fit in the cache as long as the subscription workload contains only on the order of a few million of distinct predicates, for which only a few mega bytes of cache is required\(^1\). However, the potential source of cache misses is not limited to Result Bit-array accesses, in fact, another BE-Tree internal data structure that generates cache misses (in addition, to general pointer chasing of any tree structure) is the representation of a leaf node’s content.

In BE-Tree with bitmap-based evaluation, since each subscription in the leaf node requires only to keep an array of references to Result Bit-array, it is feasible to store all subscriptions in the leaf node as cache-conscious blocks of 2-dimensional arrays of references. Moreover, since in BE-Tree, the number of subscriptions in each leaf node is limited to only tens or hundreds of subscriptions [176, 179], the 2-dimensional subscription representation could also be fitted in the processor cache; thus, substantially

\(^1\)The processor used in our experiment has two shared-cache blocks of size 6144KB.
reducing the number of cache misses during subscription evaluations at the leaf level improving the overall matching time.

4.3 Matching on Compressed Events

By exploiting the bitmap-based encoding from Section 3.7, in this chapter, we develop a novel parallel matching algorithm (PCM) that carries out subscription matching in parallel over compressed events. The compressed events are created by coalescing multiple bitmap-based event encodings into one. Our event compression technique, unlike convex-shaped minimum bounding area techniques, prevalent in the database multi-dimensional indexing literature [87], geometrically represents many events using a non-convex minimum bounding area that avoids including in the bounding area any empty space, i.e., dead space, as shown in Figure 4.2. Inclusion of dead space results in increased chance of false candidates.²

4.3.1 Parallel Compressed Matching

In a sequential processing setting, as an event arrives, we first compute a bitmap-based encoding of the event, then we use the bitmap encoding for traversing BE-Tree and finding the match results. This process is illustrated in Figure 4.3.

In a parallel processing setting with compression, as part of our PCM algorithm, we leverage the bitmap-based encoding to provide an effective compression algorithm that coalesces a set of incoming events into a single representation, namely, a compressed event, and subsequently traverse BE-Tree solely using this compressed event. Not only does the compressed event inherit all desired properties of the bitmap-based encoding such as cache-consciousness and exactly-once predicate evaluation, but it also geometrically represent a set of events as a non-convex minimum bounding area with no dead space, an

²False candidates are subscriptions retrieved that are not matched by incoming events when scanning BE-Tree leaf nodes.
The efficient construction of a compressed event is achieved by a bitwise-OR operations over the bitmap-based encoding of a set of events. The compressed event is then used to traverse BE-Tree once for all compressed events and identify all leaf nodes with potential matching subscriptions. In the final stage of PCM, we determine for each event the actual matched subscriptions using the individual bitmap-based encoding for each event. The five stages of the PCM algorithm for matching over compressed events is depicted in Figure 4.4.

Most notably, the PCM algorithm can essentially solve the matching problem for a set of events with a single BE-Tree traversal and with a single pass over all relevant leaf nodes. Thus, our approach amortizes the memory accesses and matching cost over a set of events exhibiting a substantial improvement in matching throughput (and arguably in matching latency).

The PCM algorithm is also amenable to a high degree of parallelization. In particular, Stage 1 of our algorithm (as shown in Figure 4.4) can be implemented using $n$ threads for $n$ events such that each thread is responsible to compute the bitmap-based encoding for each event. Similarly, Stage 2 coalesces the bitmap encoding of $n$ events into a single compressed event, which again can be done in parallel if the bitwise-OR computation is horizontally partitioned across $n$ threads. Let the size of the bitmap encoding be given by $\text{length}(\text{bitmap})$ bits, then the bitmap-based encoding is horizontally partitioned into chunks such that each thread is assigned one chunk, and a chunk is $\text{length}(\text{bitmap})/n$ number of consecutive bits (Stage 3). Furthermore, the chunk size is adjusted to be a multiple of a cache-line size in order to avoid false sharing, i.e., false sharing occurs when in a shared-memory multi-threaded program, threads’ local objects fall within the same cache-line, thereby unnecessarily triggering a cache-coherency protocol. False sharing is known to substantially reduce the overall performance of multi-threaded applications [134].

The resulting compressed bitmap encoding of all events (the compressed event) is then passed onto BE-Tree (Stage 4). During the matching process (within a single event) all matching candidate subscriptions
for the compressed event are collected upon reaching the leaf nodes in BE-Tree. Finally, in Stage 5 of the PCM algorithm, within each reachable BE-Tree leaf node, every subscription residing in the leaf node is matched individually for each event (using its pre-calculated bitmap encoding in Stage 1). Therefore, at this point, all the matching subscriptions can be determined exactly for each event. In addition, each event can carry out the final check within a single thread. Most importantly, all parallelized stages of our PCM algorithm are independent within each stage and require no coordination, which again improves running time considerably. In addition, only limited overall coordination and no data consistency is required across threads. In fact, the only coordination requirement is the in-order processing of stages: Each thread in a stage must wait until all running threads have completed before proceeding to the next stage.

Lastly, it is important to note when matching with compressed events using the bitmap-based encoding (Stages 4-5), that all predicate evaluations including predicates associated with BE-Tree’s c-nodes, are simply translated into direct lookup in the Result Bit-array. As a result, eliminating the need for re-evaluating any predicates for any of the compressed events. The BE-Tree extension to support our PCM algorithm is demonstrated in Figure 4.5.

4.3.2 Traversal Unrolling

The first PCM algorithm optimization is the dynamic loop unrolling (i.e., loop unwinding) for BE-Tree traversal. In the base PCM algorithm following the BE-Tree algorithm [176], the tree is traversed recursively (resulting in a depth-first search), which continuously switches between leaf node scanning (and parallel matching) and traversing the remainder of the tree. This constant switching introduces additional overhead every time leaf node scanning is initiated: (1) events-to-thread assignment, (2) threads-to-processor or threads-to-core assignment, and (3) thread creation and destruction depending on the thread implementation.³

As discussed previously, although in theory, every thread is run independently of other threads, the

³For instance, in certain Open Multiprocessing libraries, the programmer has no direct access to control the underlying thread creation/ destruction.
re-assignment, that occurs after every leaf node scanning, may reshuffle threads and core assignments causing false sharing. For example, each thread keeps track of a number (and actual) of matches for each event and the thread’s local variables are brought into non-shared L1 cache of the core that is running the thread. Now suppose, when the first leaf node is examined, the event $e_i$ is assigned to the thread $t_j$ running on the core $c_k$, but upon a subsequent assignment, suppose that the thread assignment is re-shuffled such that $t_j$ is now assigned to $c_{k+1}$ and $t_j$ continues to be responsible for the event $e_i$. Thus, the $e_i$ counter is brought into the non-shared L1 cache of both $c_k$ and $c_{k+1}$. Now, as soon as the $e_i$ matching counter is incremented by $t_j$, an expensive cache coherency protocol is triggered, invalidating and synchronizing $c_k$’s local cache despite the fact that $c_k$ will never be read and could have simply been ignored (i.e., an instance of false sharing).

The three issues raised above can be addressed, if tree traversal and leaf scanning steps are de-coupled, conceptually, resulting in a dynamic tree traversal unrolling. Thus, we introduce traversal unrolling in PCM as follows. First, traverse BE-Tree and collect all candidate leaf nodes (stored in a form of list-based structure), and, second, we iterate through all leaf nodes and scan their contents. In this way, the threads-to-processor and threads-to-core assignments occur only once, which eliminates the problem of false sharing and cache coherency all together. Furthermore, unrolling also avoids unnecessary thread creation and destruction.

Moreover, if the list-based structure for holding candidate leaf nodes is replaced with a max-heap structure, then the top-k processing proposed in [178] is also directly applicable. Therefore, the order in which leaf nodes are examined is determined by the top-k scoring and the max-heap always provides constant time access to the most relevant leaf nodes.

### 4.3.3 Parallel Path Traversal

The next PCM algorithm optimization is the acceleration of the matching computation by parallelizing the BE-Tree traversal. In which conceptually BE-Tree is divided into sets of regions, and each region is assigned to one thread. Each region can be defined as a set of paths that needs to be examined. Recall that a BE-Tree computation (an event matching operation) requires a multi-path traversal. To form these regions, different scheduling schemes for paths-to-thread assignment are possible.

Under the static assignment, a round-robin assignment of paths to threads has proven to be effective in our evaluation (and is a default option in all of our experiments). One possible shortcoming of a round-robin assignment is underutilization of processing power for certain workload distributions such that for a given event certain threads remain idle because their assigned region has no path that is relevant for the given event(s). To the contrary, the dynamic assignment relies on the paths-to-thread assignment at run-time after identifying potential paths for a given (compressed) event.

Note that the parallel path traversal optimization does not change the tree traversal and leaf scanning de-coupling of the PCM algorithm that we introduced in the last section.

### 4.3.4 Matching Pipeline

A natural pipeline arises in our PCM proposed compressed matching computation, namely, bitmap-based event encoding, event stream compression, and event matching. This pipeline can be mapped onto the symmetric multiprocessing (SMP) architecture that connects a set of identical processors using shared main memory (spread over several memory banks all connected by a system bus.) The main challenges
that arise in exploiting an SMP is limited data bandwidth with respect to the available processing bandwidth. In order to avoid any processor idleness, one must ensure that processors are fed directly by their closest memory banks, coupling memory and processor. We achieve memory and processor coupling by relying on non-uniform memory access (NUMA) and by replicating BE-Tree in the processor’s local memory banks.\(^4\)

### 4.3.5 Parallel Implementation

We use the portable Open Multiprocessing (OpenMP) library for implementing our PCM algorithm including our novel compression, traversal unrolling, and parallel path traversal techniques. In addition, OpenMP provides both static and dynamic thread scheduling as a tunable parameter, which we utilized in our evaluation.

### 4.4 Online Event Stream Re-ordering

To realize the true power of the PCM algorithm’s compressed matching, the compressed events must be similar. A naive solution would simply rely on the event order, which may fluctuate and contain random noise (i.e., events), that deteriorates the effectiveness of the compression technique. Therefore, it is essential to account for noise in the event stream and proactively bring together similar events that are close to each other but not adjacent. This goal is achieved by our online stream re-ordering technique (OSR).

Unlike the traditional online re-ordering algorithms that rely on variation of locality-sensitive hashing, we argue that although these approaches are suitable for re-ordering of data over high-dimensional space (similar to our event space), yet these approaches are inadequate in our setting. Basically, any online re-ordering algorithm that relies on locality-sensitive hashing can assign a set of events into a set of clusters (i.e., bucket), where the number of clusters is much smaller than the number of events). Ideally, each cluster will contain similar events. Such a re-ordering algorithm conceptually resembles approximate sorting.

However, there are four major shortcomings of any algorithm that relies solely on locality-sensitive hashing. First, locality-sensitive hashing requires tuning many parameters, a non-trivial and daunting task \[69\]. Second, there is no efficient technique for reasoning about the similarity among events in each cluster. We refer to this as discovering the degree of stream heterogeneity. Third, more importantly, there is no way for controlling the size of these clusters, i.e., the distribution of events across clusters maybe highly skewed. Forth, in locality-sensitive hashing, each hashed data item over the high-dimensional space is assumed to be fully defined over the entire space. However, in our domain, most events, specify values for only small sets of dimensions and provide no values for the remaining dimensions. We refer to this irregularity as incomplete event data.

To address these shortcomings, we propose a new online re-ordering technique OSR that solves all the above-mentioned shortcomings. Notably, our online re-ordering exploits BE-Tree itself to re-order the event stream. As discussed before, BE-Tree is one of the best known algorithms for indexing Boolean expressions over a high-dimensional space \[176\], which is also one of the strength of locality-sensitive hashing.

\(^4\)Using NUMA memory-processor coupling, we have achieved linear scaling of the PCM algorithm when scaling to eight processors of an 8-core Intel Xeon X6550.
In short, the PCM algorithm operates as follow. In order to re-order the event stream, events are buffered into batches (of size $b$), then the batched events are inserted into BE-Tree. The resulting tree is a set of clusters that hold similar events together. All events in each event cluster are then compressed and matched in turn.

Since BE-Tree takes as inputs the minimum and maximum cluster size, then by design, BE-Tree offers control over the size of each cluster. In particular, it is desirable that the minimum size of each event cluster should be larger than the number of available threads in order to achieve the maximum benefit and avoid thread idleness.\footnote{The effect of the event cluster size is studied in Section 4.5.2.} In addition, BE-Tree also supports incomplete event data by design [176]. More importantly, the heterogeneity of each cluster can efficiently be derived based on the depth of any cluster in the BE-Tree. For example, all events assigned to the root cluster do not have predicates on any common attributes, or all events in a cluster at depth two of the tree, have predicates defined on at least two common attributes, where the range values defined by these predicates are also overlapping.

Exploiting the most important feature of the OSR algorithm is the ability to reason about stream heterogeneity and dynamically adapt to similarity among events in the stream. Therefore, we proposed an adaptive parallel compressed matching algorithm (A-PCM) that utilizes both the parallel compressed matching algorithm and the standard parallel matching algorithm.\footnote{For the standard parallel algorithm, we process $n$ events in parallel using $n$ threads, where each event is assigned one thread and all threads operate independently in complete isolation [76].}

Our A-PCM algorithm works as follows. As the stream is batched and re-ordered on-the-fly using the OSR technique, for each batch of events, all event clusters below a certain similarity threshold are processed as uncompressed using the standard parallel matching algorithm while event clusters above a certain similarity threshold are processed using the compressed matching technique. All events in each cluster that satisfy the threshold condition are compressed together. Consequently, the number of compressed clusters of events are proportional to the number of clusters that are above the similarity threshold. The A-PCM algorithm is depicted in Figure 4.6.
4.5 Evaluations

We present a comprehensive evaluation of our PCM, A-PCM, and OSR algorithms using both synthetic and real datasets. The experiments were conducted on a machine with two Quad-core Intel Xeon X5450 processors running at 3.00 GHz with two 6MB of shared L2 cache and 16GB of memory. All algorithms are implemented in C (compiled with version gcc 4.1.2 and O3 optimization level) using OpenMP 2.2 and are extensions of the BE-Tree 1.3 open source project.\(^7\)

4.5.1 Experiment Overview

First, we demonstrate the effectiveness of the PCM optimizations and the importance of the A-PCM algorithm. Second, we compare our adaptive A-PCM algorithm with BE-Tree, which is known to be one of the fastest matching algorithm [176], under controlled experimental conditions, before showing results on real-world data. All workload are generated using the open source Boolean expression generator, BEGen\(^8\) [176]. In particular, in our evaluation, we varied workload distribution, workload size, space dimensionality, event matching probability, and, most importantly, event stream similarity.

The value of each parameter in our synthetic and real-world data\(^9\) workloads are summarized in Table 4.1 (Columns 1-5), in which each column corresponds to a different workload profile while each row corresponds to the actual value of various workload parameters. All workloads are generated by BEGen.

In our evaluation, we generate workloads having a controlled degree of event matching probability, which ensures that each event in the event workload matches a certain percentage of all subscriptions. For example, the matching probability of \(m\)% means that each event matches at least \(m\)% of all subscriptions. Similarity, BEGen enables controlling the event stream similarity. For instance, an event workload with stream similarity of \(s\)% means that each event has been replicated \(k\) times using a Gaussian distribution, in which \((1−s)\)% of predicates in the replicated events have been replaced using new random predicates, e.g., if \(s = 100\)%, then the \(k\) replicated events are all identical, and if \(s = 0\)%, then all the \(k\) replicated events are completely different. As a result, if the initial event workload has a matching probability of \(m\)%, then by applying the stream similarity technique, the final event workload may exhibit on average a lower event matching probability.

In our micro experiments, we individually study the effectiveness of each optimization in the PCM algorithm, namely, the unoptimized parallel compressed matching algorithm (C), the parallel compressed algorithm with traversal unrolling (C-U), the parallel compressed algorithm with traversal unrolling and parallel path processing (C-U-PP), the parallel compressed algorithm with traversal unrolling, parallel path processing combined with the OSR technique (C-U-PP-RE), and, finally, our “flagship” algorithm, the adaptive parallel compressed algorithm with traversal unrolling, parallel path processing and OSR technique (A-PCM). Also, in all our experiments, the matching time for C-U-PP-RE and A-PCM also includes the time taken for online stream re-ordering.

In our macro experiments, we consider (1) BE-Tree [176], (2) BE-Tree with a bitmap-based encoding (Bitmap) [179], (3) base parallel BE-Tree, which simply process \(n\) events over \(n\) threads in parallel (Parallel) [76], (4) the A-PCM algorithm, where the stream batch size for re-ordering is set to 1024, the event cluster size for stream re-ordering is set to 8, and the stream similarity threshold is set to 2 (A-PCM).

---

\(^7\)http://msrg.org/project/BE-Tree
\(^8\)http://msrg.org/datasets/BEGen
\(^9\)BEGen is based on DBLP authors and titles data [176].
Finally, in all our experiments, the internal parameters of BE-Tree, in particular, the minimum and maximum cluster size, have been assigned based on guidelines provided in [176], e.g., the maximum cluster size is set to 5 and 20 for workloads with matching probability $m < 1\%$ and $1\% \leq m \leq 9\%$, respectively, while the minimum cluster size is fixed at 3.

### 4.5.2 Micro Experiments

We establish the importance of the PCM optimizations by varying the degree of both stream similarity and matching probability. We have experimented with both uniform and Zipf distributions and real-world data workload distributions, in which a similar overall trend has been observed in all variations. As a result, in the interest of space, we have included results for uniform distribution unless stated otherwise. The micro experiments for various optimizations are shown in Figure 4.7 and effects of internal parameters for the adaptive parallel compressed algorithm are presented in Figure 4.8.

**Effects of Traversal Unrolling.** One of the main challenges in parallelizing an algorithm is the need to reduce locking and avoid false sharing, which together could offset any potential gain resulting from parallelism [134]. The parallel compressed algorithm C has already eliminated the need for locking by assigning each event to exactly one concurrent thread. However, without fine control over event-to-thread or thread-to-core assignments, the rise of false sharing and cache coherency are inevitable due to the recursive nature of BE-Tree processing. The traversal unrolling C-U solves precisely this problem. As shown in Figure 4.7(a), the throughput has increased by 29X when applying traversal unrolling. Notably, the C algorithm throughput is actually lower than sequential BE-Tree, as shown in Figure 4.9(c), this finding confirms the claim in [134] that unless false sharing is resolved no performance gain is obtained even with a theoretically highly parallel algorithm [134].

**Effects of Parallel Path Processing.** Our second optimization is aimed to systemically eliminate the second obstacle, as formulated by Amdahl’s Law, in harnessing the true benefits of any parallel algorithm. This law states the performance gain of any parallel algorithm is limited by and inversely proportional to the time needed for the sequential fraction of the algorithm. The biggest sequential component of the C-U algorithm is traversal unrolling, which is solved by the proposed parallel path processing algorithm C-U-PP. As demonstrated in Figure 4.7(a), after parallelizing the traversal unrolling, the throughput is increased by nearly 51X over our base compressed algorithm C.
Effects of Stream Re-orderings. The last two experiments showcase the effectiveness of the parallel compressed algorithm, but they do now show the practicality and wide-applicability of our approach. The compressed algorithm C-U-PP can successfully utilize similarity in an event stream assuming that the stream is sorted in a sense that similar events are near each other (Figure 4.7(a)). But in practical setting, although there may be overlap among events, however, the events may be in any random order. Therefore, it is essential to efficiently re-order the event stream on-the-fly as achieved by our novel online stream re-ordering technique (OSR) included in C-U-PP-RE. In Figure 4.7(b), we vary stream similarity for an unsorted event stream. For C-U-PP-RE, events are buffered in batches of size one thousand (a tunable parameter), and re-ordered before passing into the parallel compressed algorithm. The re-ordering is most effective, as expected, when the stream similarity is high, in which the throughput is increased substantially by a factor of up to 38X. Naturally, as stream similarity is reduced, the benefit is also reduced. Thus, when similarity is only 10%, C-U-PP-RE remain dominant and outperforms C-U-PP by 8X.

Effects of Stream Adaptive Processing. To further exploit the efficient online stream re-ordering included in C-U-PP-RE, the proposed adaptive algorithm A-PCM also recognizes when the parallel compressed algorithm C-U-PP-RE is most effective and when the parallel BE-Tree algorithm, Parallel, is most suitable. This process interweaves with our stream re-ordering algorithm, which clusters events based on their similarities. Thus, all clusters below a certain tunable similarity threshold (2 in our case) is processed using Parallel and others are processed by C-U-PP. This adaptive processing of the event stream amounts to the substantial gain of up to 8X, as shown in Figure 4.7(c). As expected, the A-PCM algorithm is most effective when the variance in the event stream is high, which is observed when stream similarity reaches 0%. However, when the event stream is at 100% similarity (arguably an unlikely practical scenario), then, as expected, the C-U-PP-RE algorithm has slightly higher throughput than the A-PCM algorithm.

Effects of Adaptive Processing Parameters. The three A-PCM parameters that were assumed in the previous experiments are: The event stream batch size (in stream re-ordering), event cluster size (in stream re-ordering), and the similarity threshold. We observe that as the stream batch size increases, our re-ordering algorithm is able to find a better clustering, thus, resulting in an improved throughput, as shown in Figure 4.8(a). In all of our experiments, we fixed our batch size to 1024.

Unlike the effect of the batch size that was rather subtle, we observed that the event cluster size for

---

10In fact, all generated workloads except those in Figure 4.7(a), follow a random order.
stream re-ordering is critical in tuning the A-PCM algorithm. As shown in Figure 4.8(b), as we vary the stream similarity, different values of the event cluster size substantially affect the overall throughput. For example, for the lowest stream similarity, as the event cluster size is decreased from 128 to 8, the throughput is increased by a factor of 9X. This increase in throughput is justified because forcing a large event cluster size over a stream with low similarity threshold results in clusters with many unrelated events; thus, increasing the number of search paths and increasing the false candidate rate. To the contrary, we observed in all our experiments that when the cluster event size is set to the number of available threads (8 in our setting), regardless of the stream similarity threshold, the A-PCM algorithm always achieves the highest throughput. Therefore, simplifying the parameter tuning of the re-ordering technique and demonstrating that our re-ordering technique is robust with respect to the degree of the stream similarity.

The re-ordering technique is robust because when using BE-Tree for clustering events, BE-Tree automatically adapts the cluster size given the workload, and the value of the event cluster size is simply taken as an initial value. This parameter is tuned adaptively based on the size of the discovered clusters. This fact is also evident in Figure 4.8(b), which shows that the throughput of lower and higher cluster values converges.

The results for varying the similarity threshold parameter are shown in Figure 4.8(c), which is the reminiscence of the pattern observed in Figure 4.7(c). The higher the similarity threshold (and the A-PCM algorithm itself) is, the better suited it is for streams with a lower overlap among events. By using higher a similarity threshold, we ensure that only events with predicates on many common attributes are compressed together. Overall, we observe that setting similarity threshold at around 2 is most robust to workloads with different degrees of stream similarity. Thus, in all of our experiments, we choose similarity threshold of 2.

**Matching Time Breakdown.** We conclude our micro experiments by analyzing the A-PCM matching time for our default datasets with a 1% matching probability and a 70% stream similarity, as shown in Table 4.2:

<table>
<thead>
<tr>
<th>Data Sets</th>
<th>Stream Re-ordering</th>
<th>Bitmap Encoding &amp; Compression</th>
<th>Tree Traversal</th>
<th>Leaf Scanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uni</td>
<td>2.57%</td>
<td>0.91%</td>
<td>32.08%</td>
<td>63.62%</td>
</tr>
<tr>
<td>Zipf</td>
<td>0.36%</td>
<td>0.18%</td>
<td>28.61%</td>
<td>70.64%</td>
</tr>
<tr>
<td>Author</td>
<td>2.18%</td>
<td>0.76%</td>
<td>29.53%</td>
<td>66.97%</td>
</tr>
<tr>
<td>Title</td>
<td>1.94%</td>
<td>0.22%</td>
<td>25.20%</td>
<td>75.41%</td>
</tr>
</tbody>
</table>
Figure 4.9: Comparison of sequential, parallel, and adaptive parallel compressed techniques

in Table 4.2. The overall time is broken down into three main components: Stream re-ordering, parallel bitmap encoding and compression (Stages 2-3 in Figure 4.4), parallel tree traversal (Stage 4 in Figure 4.4), and parallel leaf scanning (Stage 5 in Figure 4.4). The key finding is that the time for stream re-ordering, parallel bitmap encoding, and compression is negligible, and on average the leaf scanning takes twice as long as the tree traversal. It is noteworthy that the ratio between tree traversal and leaf scanning may vary depending on the workload matching probability.

4.5.3 Macro Experiments

After incrementally showing the effect of each of the proposed optimizations over the (adaptive) parallel compressed algorithm, we conclude that the A-PCM algorithm is most effective. Next, we compare A-PCM to other sequential, parallel, and adaptive parallel compressed techniques.
PCM with the best known sequential matching algorithm, BE-Tree [176], and its straightforward parallel counterpart, Parallel; these simple matching algorithms parallelization were also suggested in [76].

**Effects of Workload Size.** As we increase the number of subscriptions from 0.5M to 5M, the gap between the A-PCM algorithm and BE-Tree substantially increases, 153X and 217X increase in throughput when having only eight parallel threads. The bitmap-based encoding of BE-Tree also results in an improvement over BE-Tree (e.g., up to 12X). Most striking is the improvement of our A-PCM over Parallel, an increase of up to 10X. The gap further widens when scaling the workload size. Finally, we observe that the A-PCM algorithm can sustain a rate of up to 17,947 events/second when there are as many as half a million subscriptions, as shown in Figure 4.9(a).

**Effects of Workload Dimensionality.** Another key workload characteristic is dimensionality (Figure 4.9(b)). In fact, the curse of dimensionality is the most challenging aspect of our problem. Similar to BE-Tree, the A-PCM algorithm copes with a space dimensionality in the hundreds, which is partly due to the effective online stream re-ordering that is robust with respect to both stream incompleteness and high-dimensionality. The A-PCM algorithm continues to outperform both BE-Tree and Parallel algorithms by up to 162X and 7.7X, respectively.

**Effects of Matching Probability.** Another key distinguishing workload property is the matching probability, which assess the applicability of matching algorithms for a wide-range of applications faring anywhere from only a few matches to hundreds of thousands of matches per event. In all experiments for both synthetic and real workloads, The A-PCM algorithm remain dominant regardless of matching probability and outperforms the Parallel algorithm by up to 27X and 3.9X, when the matching probability is 0.01% and 9%, respectively, as shown in Figures 4.9(c)-4.9(d). Furthermore, the A-PCM algorithm substantially improves over sequential BE-Tree by a factor of 503X.

**Effects of Stream Similarity.** This is one of our most important experiment, which establishes that the A-PCM algorithm can take full advantage of stream similarity even when the stream is unsorted. Thus, the A-PCM algorithm re-orders and identifies similar events, compresses similar events, and process the compressed stream in parallel. In Figure 4.10(a), as we vary stream similarity (starting with the initial matching probability of 1%), our A-PCM algorithm outperforms BE-Tree and the Parallel algorithm by 330X and 19X, respectively.

From an experimentation point of view, as discussed previously, a side-effect of varying stream similarity is the loss of control over the event matching probability, which explains why for a lower stream similarity, the throughput of all algorithms are higher. In order to keep the matching probability constant, we repeat the same experiments but without controlling the matching probability, cf. Figure 4.11(a). As expected, we observe that varying the stream similarity has no effect on the BE-Tree, Bitmap, and Parallel algorithms, but A-PCM is substantially improved by up to 5.9X for an increasing degree of stream similarity. As a result, A-PCM can sustain an event rate of up to 233,863 events/second while the Parallel algorithm is saturated with only 1,429 events/second, over two orders of magnitude slower.

**Effects of Number of Distinct Predicates.** In all the above experiments, we kept the average number of distinct predicates below a million. Next, we present the result of increasing the number of distinct predicates to tens of millions, such that even a single bitmap-based encoding will not fit in the processor caches entirely. In Figure 4.10, we capture the effects of scaling the number of distinct predicates from tens of thousands to tens of millions. As expected, our A-PCM algorithm is most effective when the bitmap-based event encoding of all compressed events fits into the processor cache, to be more
Effects of Event Batching on Latency. Our second most important experiments is focused on average matching latency. Hitherto, we demonstrated that A-PCM could substantially improve the matching throughput when batching events, but we have ignored the effects of batching events on the latency.

We now consider a scenario in which we decrease the average delay latency between events (i.e., increasing the event rate) from 100ms to 0ms, as shown in Figure 4.11. For this experiment, we focus our study only to the fastest sequential (Bitmap) and parallel (A-PCM) algorithms observed in our evaluation. Furthermore, we construct the best possible scenario for the latency computation of our Bitmap technique, in which the experiment is repeated for every batch of 128 events implying that after every 128 events, we reset all counters and starts afresh.12 This method is specially advantageous (substantially underestimating the true latency) for a high-throughput event stream when using Bitmap because we ignore the fact that over time, the backlog of unprocessed events (pending queued events) continues to grow, which in practice will further increases the average matching latency. But even in such a biased experiment setting towards our Bitmap technique, we establish the superiority of the A-PCM algorithm.

We draw two key observations. First, as long as the average matching latency is smaller than event

---

12 Following the same methodology, we reset counters for A-PCM after processing a batch of X events, where X is chosen from 128, 256, 512, or 1024 events.
delays, then clearly any sequential algorithm will outperform, with respect to the average latency, any parallel algorithm that relies on batching because in the time that requires to batch a set of events, the entire matching computation could have been completed. This breakaway point in our experiments is when the average delay between events are smaller than 1-10ms.\textsuperscript{13} Second, most importantly, when the average delay between event is about a factor of 10-100 smaller than matching time, then we observe that A-PCM not only substantially improve the throughput, but in fact, it reduces the overall matching latency compared to our fastest sequential sequential algorithm. As shown in Figure 4.11, when the stream rate is higher than 10,000 events/second, then A-PCM exhibits a lower matching latency compared to Bitmap sequential algorithm. Thereby, making A-PCM a robust matching algorithm with respect to both throughput and latency dimensions for high-throughput event stream.

**Effects of CPU Performance Counter.** To substantiate our analytical claims that our proposed A-PCM is both cache-friendly (reducing cache-misses) and algorithmically efficient (reducing the number of instructions), we extract performance counter by capturing CPU low-level hardware events using the Linux profiler tool called `perf`.\textsuperscript{14} In Figure 4.12, for the same experiment, in which we vary the stream similarity, we present the average matching throughput (in Figure 4.12(a)), the raw number of instructions for an average run of our experiment (in Figure 4.12(b)), the raw number of cache-misses for an average run of our experiment (in Figure 4.12(c)), and the percentage of cache-misses for an average run of our experiment (in Figure 4.12(d)). To distinguish the benefits of parallelization and algorithmic efficiency of our A-PCM, we devise two versions of A-PCM: the parallel version (A-PCM) and the sequential

\footnotesize
\textsuperscript{13}In another words, when the event stream rate is below 1000 events per second.

\textsuperscript{14}For gathering the performance counter we used a newer machine with one Quad-core Intel Xeon W3565 processor running at 3.20GHz with shared 8MB L3 cache size.
version (A-PCM$_{Seq}$).

We observe that our compressed matching algorithm indeed algorithmically more efficient, i.e., partly owing to reduction of raw number of executed instructions by a factor of 0.5 in Figure 4.12(b), because A-PCM$_{Seq}$ throughput outperforms both sequential Bitmap and Parallel algorithms, as shown in Figure 4.12(a). In addition to algorithmic superiority of A-PCM and A-PCM$_{Seq}$, our compression matching algorithm exhibits a better cache-locality, which is reflected in the reduced number of caches-misses in Figure 4.12(c). In both A-PCM and A-PCM$_{Seq}$ (as expected, they exhibit almost an identical number of cache-misses), the number of cache-misses compared to Bitmap and Parallel are reduced by a factor of 2 and 8, respectively.\(^\text{15}\) Similarly, the percentage of cache-misses and inferred total number of cache-reference requests in A-PCM and A-PCM$_{Seq}$ are smaller, as shown in Figure 4.12(d). Hence, our compressed matching algorithm reduces the matching computation and improves the cache-friendliness.

### 4.6 Summary

In this chapter, we study the problem of parallel event processing. Particularly, we enhanced and parallelized an existing state-of-the-art matching algorithm by introducing a novel event stream compression algorithm enabled through a bitmap-based event encoding. Furthermore, we developed an efficient online event stream re-ordering (OSR) approach to exploit the full potential of our adaptive parallel compressed matching algorithm (A-PCM). Our extensive evaluation demonstrates the effectiveness of the proposed A-PCM algorithm that outperformed state-of-the-art sequential and naive parallel algorithms by a factor of up to 503X.

\(^{15}\)For Parallel algorithm, many cache-misses and the subsequent memory-requests may be issued by CPU in parallel; thus, the overall delay of cache-misses could be smaller.
Chapter 5

Top-k Query Indexing (Main Memory)

5.1 Introduction

The efficient processing of Boolean expressions is a key functionality required by major data intensive applications ranging from user-centric processing and personalization to real-time data analysis. Among user-centric applications, there are targeted Web advertising, online job sites, and location-based services; common to all are specifications (e.g., advertising campaigns, job profiles, service descriptions) modeled as Boolean expressions (subscriptions) and incoming user information (e.g., user profiles) modeled as events [140, 211, 82]. Among real-time data analysis applications, there are (complex) event processing, intrusion detection, and algorithmic trading; again, common among them are predefined set of patterns (e.g., attack specifications and investment strategies) modeled as Boolean expressions (subscriptions) and streams of incoming data (e.g., stock feeds) modeled as events [12, 74, 38, 211].

Unique to user-centric processing and personalization are strict requirements to determine only the most relevant content that is both user-consumable and suitable for the often limited screen real estate of client devices. Shared among user-centric processing and real-time analysis applications are two requirements: (1) scaling to millions of patterns and specifications (expressions) and (2) matching latency constraints in the sub-second range.

To put into context the importance of finding the most relevant Boolean expression, we extract real scenarios from prominent user-centric applications such as targeted Web advertising (e.g., Google, Microsoft, Yahoo!) and online job sites (e.g., Monster).

In Web advertising, a demographic targeting service specifies constraints such as $[\text{credit-score} > 650 \ (wt = 0.4), \ \text{num-visits} > 10 \ (wt = 0.2), \ \text{age} \ \text{BETWEEN} \ \{22, 36\} \ (wt = 0.4)]$ while an incoming user’s profile includes information such as $[\text{credit-score} = 732 \ (wt = 0.2), \ \text{num-visits} = 17 \ (wt = 0.2), \ \text{age} = 27 \ (wt = 0.6)]$. Thus, only the most relevant ads for the user are displayed; the ad relevance is computed based on a scoring function that takes as inputs the constraints’ weights ($wt$) assigned by user and advertiser. Similarly, in online job sites, an employer submits the job details, $[\text{category} = \text{‘green jobs’} \ (wt = 0.2), \ \text{hours/week} > 15 \ (wt = 0.5), \ \text{hourly-rate} < 45 \ (wt = 0.3)]$, and when a job seeker visits with the profile, $[\text{category} = \text{‘green jobs’} \ (wt = 0.4), \ \text{hours/week} = 20 \ (wt = 0.1), \ \text{and hourly-rate} = 30 \ (wt = 0.5)]$, she is connected to the most relevant employer [140, 211, 82]; again the relevance is
computed based on the weights of the constraints. These scenarios demand the indexing of millions of Boolean expressions while having matching latency requirements in the sub-second range to avoid the deterioration of user-perceived latency when browsing and shopping online.

In these examples, advertisers’ and employers’ needs and users’ profiles are captured using multiple attributes (i.e., dimensions in space) such as age and credit-score, in which the importance of each attribute is explicitly captured using the attributes’ weight. Additionally, each attribute is constrained to a range of values on either continuous or discrete domains using an operator. The resulting quadruple formed by an attribute, an operator, a range of values, and a predicate weight is referred to as a Boolean predicate. A conjunction of Boolean predicates, here representing both the advertiser’s and employer’s needs and the user’s profiles, constitutes a Boolean expression (BE).

In short, we extract four requirements of paramount importance from applications that rely on processing Boolean expressions. First and foremost, the index must enable top-k matching to quickly retrieve only the most relevant expressions, which is not addressed by prior Boolean expression matching approaches [12, 74, 176]. This relevance computation must be based on a generic scoring function, which is also not addressed by prior approaches that model relevance identification as part of matching [140]. Second, the index must support predicates with an expressive set of operators over a continuous domain, which, again, is not supported by prior matching approaches [74, 211, 176]. Third, the index must enable dynamic insertion and deletion of expressions, often disregarded as requirement in matching algorithm designs [12, 211]. Finally, the index must employ a dynamic structure that adapts to changing workload patterns and expression schemata, also often disregarded in matching algorithm designs [12, 211].

We designed BE*-Tree to specifically tackle these challenges. In particular, BE*-Tree is geared towards determining the most relevant patterns, in which top-k processing is treated as first class citizen. BE*-Tree introduces a hierarchical top-k processing scheme that differs from existing work that assumes a static and a flat structure [211]. Additionally, BE*-Tree achieves scalability by overcoming the curse of dimensionality through a non-rigid space-cutting technique while attaining expressiveness and dynamism without restricting the space to only a discrete and finite domain as prevalent in [211, 176]. Notably, with respect to scalability, we solve the two critical properties common to most high-dimensional indexing: (1) avoiding indexing non-empty space (cf. R-tree [99]) (2) minimizing overlap (cf. R*-tree [26]) and coverage [99]. These properties are tackled by proposing a bi-directional tree expansion: a top-down (data and space clustering) and a bottom-up (space clustering) growths process, which together enable indexing only non-empty continuous sub-spaces and adaptation to workload changes; and a splitting strategy to systematically produce and maintain overlap-free subspaces for holding expressions.

Our contributions in this chapter are four-fold:

1. We generalize the BE matching problem to subspace matching (Section 5.2),
2. We introduce BE*-Tree tailored to top-k processing for efficiently determining the most relevant matches among large numbers of potential matches of millions of subscriptions defined over thousands of dimensions (Sections 5.3, 5.5),
3. We develop a novel self-adjusting mechanism using bi-directional tree expansion that continuously adapts as the workload changes, and propose a novel overlap-free splitting strategy (Section 5.4),
4. Finally, we present a comprehensive evaluation framework that benchmarks state-of-the-art matching algorithms, specifically emphasizing top-k processing: SCAN [220], BE-Tree [176], and k-index [211] (Section 5.6).
5.2 Language and Data Model Formalism

In this section, we extend our Boolean expression formalism in Section 3.2, thereby we define our (extended) Boolean expression language and spatial event data model followed by our (top-k) matching semantics.

5.2.1 Notation

Given an \( n \)-dimensional space \( \mathbb{R}^n \), we define the projection of \( \mathbb{R}^n \) onto \( \mathbb{R}^k \) as a \( k \)-dimensional subspace, denoted by \( \pi_{d_1 \ldots d_k}(\mathbb{R}^n) = \mathbb{R}^k \), where \( \pi_{d_1 \ldots d_k} : \mathbb{R}^n \rightarrow \mathbb{R}^k \), \( k \leq n \), and each \( d_i \in \{d_1 \ldots d_k\} \) represents the \( i^{th} \) dimension in \( \mathbb{R}^k \) and corresponds to the \( j^{th} \) dimension in \( \mathbb{R}^n \); for ease of notation, we define the identity projection as \( \pi_I(\mathbb{R}^n) = \mathbb{R}^n \). In addition, we define a \( k \)-dimensional bounding box \( B^k \) over \( \mathbb{R}^k \) as

\[
B^k = [\min_1, \max_1] \times \cdots \times [\min_k, \max_k].
\] (5.1)

Let \( \xi_j(B^k) = [\min_i, \max_i] \) be the \( i^{th} \) boundary in \( B^k \) defined over the \( i^{th} \) dimension in \( \mathbb{R}^k \) which corresponds to the \( j^{th} \) dimension in \( \mathbb{R}^n \). Let \( \chi_i(B^k) \) be the center of the \( i^{th} \) boundary in \( B^k \) given by

\[
\lambda_i(B^k) = \max_i - \min_i.
\] (5.2)

Lastly, let \( \mu_i(B^k) = \min_i \) and \( M_i(B^k) = \max_i \) represent the minimum and the maximum value of the \( i^{th} \) boundary of \( B^k \), respectively.

5.2.2 Expression Language and Subspace Model

We support a rich Boolean expression language that unifies the subscription language and the event data model. This generalization gives rise to more expressive matching semantics while still encompassing the traditional pub/sub matching problem.

In our model, a Boolean expression is a conjunction of Boolean predicates. A predicate is a quadruple: an attribute that uniquely represents a dimension in \( \mathbb{R}^n \); an operator (e.g., relational operators \(<, \leq, =, \neq, \geq, >\)), set operators \((\in, \notin)\), and the SQL BETWEEN operator; a set of values (for discrete domains) or a range of values (for continuous domains); and an assigned predicate weight, denoted by \( P_{\text{attr}, \text{opt}, \text{val}, \text{wt}}(x) \) or more concisely as \( P(x) \). A predicate either accepts or rejects an input \( x \) such that \( P_{\text{attr}, \text{opt}, \text{val}, \text{wt}}(x) : x \rightarrow \{\text{True}, \text{False}\} \), where \( x \in \text{Dom}(P_{\text{attr}}) \) and \( P_{\text{attr}} \) is the predicate’s attribute. Formally, a Boolean expression \( \Omega \) is defined over \( \mathbb{R}^n \) as follows:

\[
\Omega = \{P_{\text{attr}, \text{opt}, \text{val}, \text{wt}}(x) \land \cdots \land P_{\text{attr}, \text{opt}, \text{val}, \text{wt}}(x)\},
\]

where \( k \leq n; \ i, j \leq k \), \( P_{i}^{\text{attr}} = P_{j}^{\text{attr}} \) iff \( i = j \).

(5.3)

We extended the semantics of projection to a Boolean expression in order to enable projecting out predicates associated with certain dimensions, i.e., \( \pi_{d_1 \ldots d_k}(\Omega) = \Omega' \), where \( h \leq k \) and \( \Omega \) is defined over \( \mathbb{R}^k \) while \( \Omega' \) is defined over \( \mathbb{R}^h \).
Chapter 5. Top-k Query Indexing (Main Memory)

The Boolean expression \( \Omega \) is said to have size \( k \), hence, \( \Omega \) is represented by \( \Gamma \)

**5.2.3 Top-k Matching Semantics**

Our formulation of subscriptions and events as expressions enables us to support a wide range of matching semantics, including the classical pub/sub matching: *Given an event \( \omega \) and a set of subscriptions \( \Omega \), find all subscriptions \( \Omega_i \in \Omega \) that are satisfied by \( \omega \).* We refer to this problem as the stabbing subscription.\(^2\)

### Table 5.1: Predicate Mapping

| \( \text{attr} \leq v_1 \) | \( (-\infty, v_1] \) |
| \( \text{attr} \leq v_1 \) | \( (-\infty, v_1] \) |
| \( \text{attr} = v_1 \) | \( [v_1, v_1] \) |
| \( \text{attr} \neq v_1 \) | \( (-\infty, \infty) \) |
| \( \text{attr} \geq v_1 \) | \( [v_1, \infty) \) |
| \( \text{attr} > v_1 \) | \( [v_1, \infty) \) |
| \( \text{attr} \in \{v_1, \ldots, v_m\} \) | \( [v_1, v_m] \) |
| \( \text{attr} \notin \{v_1, \ldots, v_m\} \) | \( (-\infty, \infty) \) |
| \( \text{attr} \) BETWEEN \( v_1, v_2 \) | \( [v_1, v_2] \) |

\[
\pi_{d_1 \cdots d_k}(\Omega) = \{\Omega' | \Omega' = \bigwedge_{i=1}^{n} \text{attr}.\text{opt. val. wt}(x) \land \cdots \land \text{attr}.\text{opt. val. wt}(x),
\forall d_i \in \{d_1 \cdots d_k\}, \text{attr}.\text{opt} = d_i, P_i(x) \in \Omega\}.
\]

Now, we are in a position to formalize the predicate mapping\(^1\) and, ultimately, to (approximately) represent an expression as a \( k \)-dimensional bounding box.

A predicate \( \text{attr}.\text{opt. val. wt}(x) \) is mapped into a 1-dimensional bounding box, denoted by

\[
\gamma(\text{attr}.\text{opt. val. wt}(x)) = B^1,
\]

as shown in Table 5.1, where \( \epsilon \) is the machine’s epsilon. Therefore, a predicate \( P(x) \) is covered by \( B^1 = [\min_1, \max_1] \) only if the permitted values defined by \( P(x) \) lie in [\( \min_1, \max_1 \)]; for brevity, we say the predicate is enclosed by \( B^1 \). Similarly, we say an expression \( \Omega \) over \( \mathbb{R}^n \) is partially enclosed by \( B^h \) w.r.t. the projection \( \pi_{d_1 \cdots d_k} \), denoted by \( \Gamma^x(\Omega) \).

\[
\Gamma^x(\Omega) = \{B^h | \forall P_i . \text{attr}.\text{opt. val. wt}(x) \in \pi(\Omega), \gamma(P_i(x)) \leq \gamma(P_i(x)) \}.
\]

Furthermore, we say an expression \( \Omega \) is fully enclosed by \( B^k \) when the identity projection \( \pi_X \) is applied, \( \Gamma^x(\Omega) = B^k \). Lastly, the smallest \( B^k \), or the minimum bounding box (MBB), that (partially) encloses an expression \( \Omega \) is given by

\[
\Gamma^x_{\min}(\Omega) = \{\arg \min_{B^k} \Gamma^x(\Omega) = B^k\}.
\]

The Boolean expression \( \Omega \) is said to have size \( k \), denoted by \( |\Omega| = k \), when having \( k \) predicates; hence, \( \Omega \) is represented by \( \Gamma^x_{\min}(\Omega) \) defined over a \( k \)-dimensional subspace.

\(^1\)The mapping strategy for predicates with operator \( \in, \notin, \neq \) is especially effective because the number of predicates per expression is on the order of tens while the number of space dimensions is on the order of thousands.

\(^2\)This is a relaxation of the stabbing query problem, in which interval cutting is generalized to subspace cutting.
SQ(\omega) and formalize it as follows:

\[
\text{SQ}(\omega) = \{ \Omega_i \mid \forall P^\text{attr, opt, val, wt}_q(x) \in \Omega_i, \exists P^\text{attr, opt, val, wt}_o(x) \in \omega, P^\text{attr}_q = P^\text{attr}_o, \exists x \in \text{Dom}(P^\text{attr}_q), P_q(x) \land P_o(x) \}. \tag{5.8}
\]

Alternatively, we can (approximately) express stabbing subscription as subspace matching in \(\mathbb{R}^n\) as follows, where the approximation is due to the mapping function \(\gamma\):

\[
\text{SQ}(\omega) = \{ \Omega_i \mid \forall P^\text{attr, opt, val, wt}_q(x) \in \Omega_i, \exists P^\text{attr, opt, val, wt}_o(x) \in \omega, P^\text{attr}_q = P^\text{attr}_o, \gamma(P_q(x)) \cap \gamma(P_o(x)) \neq \emptyset \}. \tag{5.9}
\]

Finally, we adopt the popular vector space scoring used in information retrieval (IR) systems\(^3\) for computing the score of a matched subscription \(\Omega_i\) for a given event \(\omega\) (to enable top-k computation), denoted by \(\text{score}(\omega, \Omega_i)\), and defined by

\[
\text{score}(\omega, \Omega_i) = \sum_{P_q(x) \in \Omega_i, P_o(x) \in \omega, P^\text{attr}_q = P^\text{attr}_o} P^\text{wt}_q \times P^\text{wt}_o. \tag{5.10}
\]

Similarly, we compute the upper bound score for an event \(\omega\) w.r.t. an upper bound weight-summary \((\text{sum}^\Omega\text{wt})\) for a set of subscriptions \(\Omega\) as follows

\[
\text{uscore}(\omega, \text{sum}^\Omega\text{wt}) = \sum_{P_o(x) \in \omega} P^\text{wt}_o \times \text{sum}^\Omega\text{wt}(P^\text{attr}_o), \tag{5.11}
\]

where \(\text{sum}^\Omega\text{wt}(\text{attr})\) returns the upper bound score of \(\text{attr}\) over the set of subscriptions \(\Omega\) which is given by

\[
\text{sum}^\Omega\text{wt}(\text{attr}) = \max_{\Omega_i \in \Omega, P_q(x) \in \Omega_i, P^\text{attr}_q = \text{attr}} P^\text{wt}_q. \tag{5.12}
\]

### 5.3 BE*-Tree Structural Properties

BE*-Tree is a generic index structure for indexing a large collection of Boolean expressions (i.e., subscriptions) and for efficient retrieval of the most relevant matching expressions given a stream of incoming expressions (i.e., events). BE*-Tree supports Boolean expressions with an expressive set of operators defined over a high-dimensional continuous space. BE*-Tree copes with the curse of dimensionality challenge through a non-rigid two-phase space-cutting technique that significantly reduces the complexity and the level of uncertainty of choosing an effective criterion to recursively cut the space and that identifies highly dense subspaces. The two phases BE*-Tree employs are: (1) partitioning which is the global structuring to determine the next best attribute \(\text{attr}_i\) (i.e., the \(i^{th}\) dimension in \(\mathbb{R}^n\)) for splitting the space and (2) non-rigid clustering which is the local structuring for each partition to determine the best

\(^3\) We are not limited to IR scoring, but support any monotonic scoring function.
grouping of expressions w.r.t. the expressions’ range of values for attr. BE*-Tree not only supports dynamic insertion and deletion of expressions, but it also adapts to workload changes by incorporating top-down (data and space clustering) and bottom-up (space clustering) expansion within each clustering phase.

The data clustering aims to avoid indexing empty space and to adapt to a skewed workload while space clustering aims to avoid degeneration of the structure in the presence of frequent insertions and deletions of expressions. Conceptually, the space and the data clustering techniques are a hybrid scheme that takes the best of both worlds. On the one hand, the data clustering employs data dependent grouping of expressions to adapt to different workload distributions and to avoid indexing empty space, and on the other hand, the space clustering employs space dependent grouping of expressions to accommodate an insertion-independent mechanism.

In general, BE*-Tree is an n-ary tree structure in which a leaf node contains the actual data (expressions) and an internal node contains partial information about data (e.g., an attribute and a range of values) in its descendant leaf nodes. BE*-Tree consists of three classes of nodes: p-node (partition node) for storing the partitioning information, i.e., an attribute; c-node (cluster node) for storing the clustering information, i.e., a range of values; and l-node (leaf node), being at the lowest level of the tree, for storing the actual data. Moreover, p-nodes and c-nodes are logically organized in a special directory structure for fast tree traversal and search space pruning. Thus, a set of p-nodes are organized in a p-directory (partition directory), and a set of c-nodes are organized in a p-directory (cluster directory). The overall BE*-Tree structure together with its structural properties is depicted in Figure 5.1.

5.3.1 Non-rigid Space-cutting

BE*-Tree’s non-rigid two-phase space-cutting, the partitioning followed by the non-rigid clustering, introduces new challenges such as how to determine the right balance between the partitioning and clustering, and how to develop a robust principle to alternate between both. In BE*-Tree, we propose a splitting policy to guide the non-rigid clustering phase for establishing not only a robust principle for alternating between partitioning and clustering but also for naturally adapting to the workload distributions. We begin discussing the overall structure and the two-phase cutting dynamics of BE*-Tree before presenting the key design principles behind BE*-Tree.
In BE*-Tree, the partitioning phase, conceptually a global adjusting mechanism, is the first phase of our space-cutting technique. The partitioning phase is invoked once a leaf node overflows (initially occurs at the root level). This phase involves ranking each candidate attr, attributes that appear in the expressions of the overflowed l-node, in order to determine the most effective attr for partitioning, as outlined in Algorithm 12. Essentially, this process identifies the highest ranking attr, given a scoring function (Section 5.4.2), to spread expressions into smaller groups (leaf nodes). In short, partitioning space based on a high-ranking attr enables the pruning of search space more efficiently while coping with the curse of dimensionality by considering a single attr for each partitioning phase.

Upon executing the partitioning phase, the high-dimensional indexing problem is reduced to one-dimensional interval indexing, which paves the way to exploit underlying distribution of a single attr at a time through BE*-Tree’s clustering phase, conceptually a local adjusting mechanism. At the core of our clustering phase, and ultimately of our BE*-Tree, are (1) a non-rigid clustering policy to group overlapping expressions (into buckets) that minimizes the overlap and the coverage among these buckets and (2) a robust and well-defined policy to alternate between the partitioning and the clustering (summarized in Algorithm 13).

The absence of a well-defined policy gives rise to the dilemma of whether to further pursue the space clustering or to switch back to the partitioning. Furthermore, a partitioned bucket can no longer be split without suffering from the cascading split problem [84]. Thus, a clustering policy that cannot react and adapt to the insertion sequence is either prone to ineffective buckets that do not take advantage of the domain selectivity for effectively pruning the search space or is prone to suffering from substantial performance overhead due to the cascading split problem. Therefore, a practical space clustering must support dynamic insertion and deletion and must adapt to any workload distributions, yet satisfying the cascading-split-free property.

In the clustering phase, each group of expressions is referred to as a bucket. Formally, a bucket is a 1-dimensional bounding box over attr, denoted by B, and an expression Ω with predicate \( P_{attr}^{opt,val,wt}(x) \in \Omega \) is assigned to a bucket only if

\[
\text{attr} = P_j^{attr} \quad \text{and} \quad \gamma(P_j(x)) \cap \xi_{P_j^{attr}}(B) = \gamma(P_j(x)). \tag{5.13}
\]

Furthermore, a bucket has a minimum bounding box (MBB) that partially encloses all of its expressions Ω if and only if

\[
B = \bigcup_{j \in \Omega} \Gamma_{\min}^{attr,\Omega_j}.
\tag{5.14}
\]

Moreover, each bucket is associated with exactly one c-node in the BE*-Tree, which is responsible for storing and maintaining information about the bucket’s assigned expressions.

### 5.3.2 Structural Adaptation

The problem of supporting dynamic insertion and deletion is further complicated due to the non-rigid and dynamic clustering introduced by BE*-Tree. In a rigid clustering that relies on deterministically dividing the space in half, a best-effort deletion strategy is adequate because the size of the clustering boundary is deterministic and fixed; thereby, removing an expression does not alter the boundary. However, the non-rigid clustering directory of BE*-Tree defines a relative (non-deterministic) bucket boundary
as minimum boundary to enclose all of the bucket’s expressions. Consequently, a simple best-effort approach for maintaining the boundary is inadequate because for a pathological case the boundary in \textsc{BE*-Tree} can unnecessarily span a large space, if an expression, defined over a large subspace, is inserted and later removed. To overcome this bucket boundary degeneration, \textsc{BE*-Tree} introduces a set of inactive dimensions which are used exclusively for efficiently keeping the boundary size up-to-date in presence of insertions and deletions. To incorporate the concept of inactive dimensions, we revise the bucket definition as follows:

A bucket at the \( k \)-th level of \textsc{BE*-Tree} is a \( k \)-dimensional bounding box, \( B^k \) over \( \mathbb{R}^k \), where \( \pi_{d_1\cdots d_k}(\mathbb{R}^n) = \mathbb{R}^k \), and each \( d_i \in \{d_1 \cdots d_k\} \) is chosen by the partitioning phase at the \( i \)-th level of \textsc{BE*-Tree}. Also, we refer to the first \( k - 1 \) dimensions of \( B^k \) as inactive while we refer to the \( k \)-th dimension (where \( d_k \in \mathbb{R}^n \)) as active dimension. Furthermore, the bucket’s partial enclosure property w.r.t. its expressions \( \Omega \) holds if and only if

\[
B^k = \bigcup_{\Omega_j \in \Omega} \Gamma_{\min \pi_{d_1\cdots d_k}}(\Omega_j). \tag{5.15}
\]

The \( k - 1 \) inactive dimensions correspond to \( k - 1 \) attributes selected by the partitioning phase on the path from the root to the current clustering directory at the \( k \)-th level of \textsc{BE*-Tree}. These inactive dimensions are utilized only during deletion while the active dimension is used for matching, insertion, and deletion. Without the notion of inactive dimension, deleting an expression results in the cascading update problem. The cascading update problem is an unpredictable chain reaction that propagates downwards in order to compute a new bucket boundary after an expression in one of the subtree’s buckets has been removed. Thus, at every level of the tree, a brute force boundary re-computation of every bucket is required. However, with inactive dimensions in place, the boundary re-computation is required only for the bucket that is associated with the leaf node from which the expression was removed (or updated). Moreover, this information needs to be only sent upward along the path to the root. Thus, at the higher levels of \textsc{BE*-Tree}, the new boundary can simply be reconstructed by considering the updated boundary information together with the boundary of unmodified buckets. The maintenance of active and inactive dimensions is handled by Algorithm 11.

### 5.3.3 \textsc{BE*-Tree} Invariance

Before stating \textsc{BE*-Tree}’s invariance and operational semantics (which is also an extension of \textsc{BE-Tree} formalism 3.3.3, we must distinguish among four bucket types: \textit{open bucket}: a bucket that is not yet partitioned; \textit{leaf bucket}: a bucket that has no children (or has not been split); \textit{atomic bucket}: a bucket that is a single-valued bucket which cannot further be split; and \textit{home bucket}: a bucket that is the smallest existing bucket that encloses the inserting expression.

The correctness of the \textsc{BE*-Tree} operational semantics is achieved based on the following three rules:

1. \textit{insertion rule}: an expression is always inserted into the smallest bucket that encloses it, while respecting the descendant-repelling property (Section 5.4.1).

2. \textit{forced split rule}: an overflowing non-leaf bucket is always split before switching back to the partitioning.

3. \textit{merge rule}: an underflowing leaf bucket is merged with its parent only if the parent is an open bucket.
Finally, the BE*-Tree correctness can be summarized as follows:

**Invariance:** Every expression \( \Omega \) is always inserted into the smallest bucket that encloses it and a non-atomic bucket is always split first before it is partitioned.

### 5.4 BE*-Tree Adaptiveness Properties

An essential property of any index structure is to dynamically adapt as the workload changes in order to prevent index deterioration. Supporting adaptiveness is central to the design of BE-Tree and has shaped every aspect of BE*-Tree. The adaptive nature of BE*-Tree is reflected in its bi-direction expansion that foresees index evolution, in its splitting strategy that improves search-space pruning w.r.t. the subscription workload, and in its proposed simplified ranking function that actively refines the space partitioning based on both subscription and event workloads. The core of BE*-Tree adaptiveness is captured by its insertion operation, given in Algorithms 9-13. Also, the adaptiveness in deletion is accomplished through maintenance of inactive dimension, which was discussed in Section 5.3.2, and through recycling nodes [176] (i.e., reinsertion policy) whose rank have dropped below certain threshold, our ranking function is defined in Section 5.4.2.

#### 5.4.1 Bi-directional Expansions

The main aim of the bi-directional expansion is to determine the right balance between a rigid clustering of the grid-based approach (space dependent) and a flexible and a dynamic clustering similar to that of an \( R \)-Tree-based approach (data dependent), while avoiding the cascading split problem. Therefore, the final objective is to enable BE*-Tree to model any data distribution (e.g., skewed), and only in the worst case degenerate to a grid-based approach which is best suited for a uniform distribution.
Top-down Expansion

The top-down expansion is part of the clustering phase in which an overflowing bucket is recursively split while keeping the bucket capacity within a threshold, denoted by $A$. There are two main types of bucket splitting: a data dependent splitting strategy (data clustering), discussed in Section 5.4.1, and a space dependent splitting strategy (space clustering) which follows a grid-based splitting, also in Section 5.4.1. As a result, whenever the data clustering approach fails to split the data into two or more buckets, while satisfying a minimum capacity requirement, denoted by $\alpha$, for each bucket, it relies on space clustering such that the bucket is split in half, as shown in Algorithm 10, Lines 11-25.

Bottom-up Expansion

The bottom-up expansion reverses the conventional tree top-down expansion. It focuses on dynamically adjusting BE*-Tree’s clustering directory to accommodate workload changes. Most importantly, the bottom-up expansion avoids BE*-Tree degeneration by preventing the partitioned bucket’s size to expand indefinitely. Therefore, to achieve this goal, the clustering directory is re-adjusted by injecting a new bucket in the upward direction, whenever, a partitioned bucket is required to host an expression which is $\beta$ times larger than the size of all expressions in its subtree. This procedure is illustrated in Figure 5.2 and its pseudocode is provided in Algorithm 10, Lines 6-10.

Descendant-repelling & Overlap-free Split

We present a novel splitting strategy for overflowing buckets, a strategy that falls in between the traditional overlapping-style splitting and grid-style rigid splitting. Our new splitting algorithm aims at systematically reducing the overlap, similar to grid-based structures, while adapting to the data distribution, similar to $R$-Tree-based structures. To achieve these properties, the overflowing bucket is split into three buckets: left, overlapping, and right buckets, such that the left and right buckets (descendant
Algorithm 9 Insert(Ω, cnode, cdir)
1: /* Find attr with max rank not yet used for partitioning*/
2: if cnode.pdir ≠ NULL then
3: for P^attr_i ∈ ω do
4: if !IsUsed(P^attr_i) then
5: pnode ← SearchPDir(P^attr_i, cnode.pdir)
6: if pnode ≠ NULL then
7: foundPartition = true;
8: if maxRank < pnode.rank then
9: maxPnode ← pnode
10: maxRank ← pnode.rank
/* if no partitioning found, then insert into the l-node*/
11: if !foundPartition then
12: Insert(Ω, cnode.lnode)
13: UpdateLNodeTopNScore(Ω, cnode.lnode.sum^Ω)
/* if c-node is the root, then do partitioning; otherwise clustering*/
14: if isRoot(cnode) then
15: Partitioning(cnode)
16: else
17: Clustering(cdir)
18: else
19: maxCdir ← InsertCDir(Ω, maxPnode.cdir)
20: insert(Ω, maxCdir.cnode, maxCdir)
21: UpdatePNodeScore(maxPnode)
22: UpdatePNodeTopNScore(Ω, maxPnode.sum^Ω)

buckets) repel each other. Although their boundary can be dynamically changed to adapt to the data distribution, they can never overlap. Consequently, data that spans over the boundary of the descendant buckets is placed in the overlapping node. Thus, by definition the overlapping bucket is the home bucket to all of its data, and upon overflowing, the overlapping bucket is handled by switching back to the partitioning phase.

An important property of the overlap-free splitting, guaranteed under the uniform distribution, is realized when a point event (i.e., consisting of only equality predicates) falls within the overlapping region. This results in a substantial reduction of the search space in which only \( \frac{2}{3} \) of the data must be scanned as opposed to checking all of the data when only two buckets are used and bucket-overlap is allowed. The \( \frac{2}{3} \)-reduction in search space is due to the proposed splitting of overflowing buckets into three buckets, a left child, a right child, and an overlapping bucket such that left and right children never intersect, which is enforced by our descendant-repelling property. This overlap-free splitting guarantees that a point-event can cut through at most one child. Thus, in the worst case only \( \frac{2}{3} \) of the data is visited.\(^4\) The resulting reduction is substantial because an entire BE*-Tree subtree could be associate with each bucket. Finally, under the descendant-repelling property, all subscriptions under any sibling buckets are guaranteed to be disjoint on at least one dimension. Next, we present two algorithms to construct descendant-repelling and overlap-free splitting (as shown in Figure 5.3).

**Exhaustive Splitting Algorithm** Our first algorithm finds an effective splitting hyperplane to divide overflowing buckets into three buckets. In the splitting of the overflowing bucket within the

\[^4\]The \( \frac{2}{3} \)-reduction can simply be applied to extended events (events with non-equality predicates), if the descendant-repelling property is generalized to enforce that sibling buckets must be separated by the average extension size of events, where the extension is defined as \( \gamma(P^attr_i, x^opt, x^val, x^wt) \).
possible splitting choices, namely, the exhaustive splitting algorithm is dominated by the sorting phase together with enumeration over all possible splitting choices, namely, $O((n - 2\alpha)n \log(n))$, where $n = |\Omega|$ and $\alpha$ is the minimum capacity of the resulting right and left buckets after splitting. In brief, the algorithm first assigns the $I$ leftmost expressions to the left bucket (or $I$ rightmost expressions to the right bucket); second, it distributes the remaining expressions to the right (or the left) and the overlapping buckets accordingly. The main objective of the algorithm is to find the optimal $I$ that minimizes overlap and coverage while satisfying the descendant-repelling property and minimum bucket capacity $\alpha$. The splitting is formally defined as follow:

**Algorithm 10** InsertCDir($\Omega, cdir$)

1. $B^k_L, B^k_O, B^k_R \leftarrow cdir, B^k_L, cdir, B^k_O, cdir, B^k_R$
2. $B^k \leftarrow B^k_L \cup B^k_O \cup B^k_R$
3. if IsLeaf($B^k_O$) then
5. return $cdir$
4. /* Bottom-up: if total $k^{th}$ boundary is expanded by more than $\beta$*/
6. else if $\lambda_k(\gamma(\pi_{d_k}(\Omega)) \cup B^k) > \beta \times \lambda_k(B^k)$ then
7. $B^k_{new} \leftarrow \Omega$
8. $cdir \leftarrow B^k_{new} \rightarrow \text{new parent} B^k_L, B^k_O, B^k_R$
9. UpdateBucketInfo($\Omega, B^k_{new}$)
10. return $cdir$
11. /* Top-down: otherwise*/
12. /* If descendant-repelling property is violated by inserting $\Omega$*/
13. if $\gamma(\pi_{d_k}(\Omega)) \cap \pi_k(B^k_L) \neq \emptyset \land \gamma(\pi_{d_k}(\Omega)) \cap \pi_k(B^k_R) \neq \emptyset$ then
14. UpdateBucketInfo($\Omega, B^k_L$)
15. return $cdir$
16. else if $\gamma(\pi_{d_k}(\Omega)) \cap \pi_k(B^k_L) \neq \emptyset$ then
17. InsertCDir($\Omega, B^k_L$)
18. else if $\gamma(\pi_{d_k}(\Omega)) \cap \pi_k(B^k_R) \neq \emptyset$ then
19. InsertCDir($\Omega, B^k_R$)
20. else
21. /* insert into a bucket which results in minimum expansion*/
22. if $\lambda_k(\gamma(\pi_{d_k}(\Omega)) \cup B^k_L) < \lambda_k(\gamma(\pi_{d_k}(\Omega)) \cup B^k_R)$ then
23. return InsertCDir($\Omega, B^k_L$)
24. else
25. return InsertCDir($\Omega, B^k_R$)

**Algorithm 11** UpdateBucketInfo($\Omega, cdir, B^k$)

1. /* Update the active dimension (the bucket $k^{th}$ dimension]*)
2. $B^k \leftarrow \gamma(\pi_{d_k}(\Omega)) \cup \pi_k(B^k)$
3. /* Update the inactive dimensions ($1 \cdots (k-1)^{th}$ dimensions]*)
4. $B^k \leftarrow \Gamma_{\text{min}, \cdots, (k-1)}^{\pi_{d_1} \cdots \pi_{(k-1)}}(\Omega) \cup \pi_{1 \cdots (k-1)}(B^k)$
4. UpdateCNodeInactiveTopNScore($\Omega_i, B^k, \text{cnode}.\text{sum}_i$)
Algorithm 12 Partitioning(cnode)
1: \textit{lnode} \leftarrow \textit{cnode}.\textit{lnode}
2: \textbf{while} \text{IsOverflowed}(\textit{lnode}) \textbf{do}
3: \hspace{1em} \textit{attr} \leftarrow \text{GetNextHighestScoreUnusedAttr(\textit{lnode})}
4: \hspace{2em} \textit{pnode} \leftarrow \text{CreatePDir(\textit{attr}, \textit{cnode}.\textit{pdir})}
5: \hspace{1em} /* \text{Create new partition for the next highest ranked attr} */
6: \hspace{2em} \text{for} \ \Omega_i \in \textit{lnode}.\Omega \text{ do}
7: \hspace{3em} \text{if} \ \exists P^\text{attr}_j \in \Omega_i, P^\text{attr}_j = \textit{attr} \text{ then}
8: \hspace{4em} \textit{cdir} \leftarrow \text{InsertCDir(\Omega_i, \textit{pnode}.\textit{cdir})}
9: \hspace{3em} \text{Move}(\Omega_i, \textit{lnode}, \textit{cdir}.\textit{cnode}.\textit{lnode})
10: \hspace{2em} \text{UpdatePNodeTopNScore(\textit{pnode}.\textit{sum}_\Omega wt)}
11: \hspace{1em} \text{Clustering(\textit{pnode}.\textit{cdir})}
12: \hspace{1em} \text{UpdateClusterCapacity(\textit{lnode})}

Algorithm 13 Clustering(cdir)
1: \textit{lnode} \leftarrow \textit{cdir}.\textit{cnode}.\textit{lnode}
2: \textbf{if} \text{!IsOverflowed(\textit{lnode})} \text{ then}
3: \hspace{1em} \text{return}
4: \textbf{if} \text{!IsLeaf(\textit{cdir}) or IsPartitioned(\textit{cdir})} \text{ then}
5: \hspace{2em} \text{Partitioning(\textit{cdir}.\textit{cnode})}
6: \textbf{else}
7: \hspace{2em} \textbf{if} \text{Exhaustive Splitting Algorithm} \text{ then}
8: \hspace{3em} \textit{cdir}.B^L_k, \textit{cdir}.B^O_k, \textit{cdir}.B^R_k \leftarrow \text{buckets}_{\text{opt}}
9: \hspace{2em} \textbf{else}
10: \hspace{3em} \textit{cdir}.B^L_k, \textit{cdir}.B^O_k, \textit{cdir}.B^R_k \leftarrow \text{buckets}_{\text{apx}}
11: \hspace{2em} \textbf{for} \ \Omega_i \in \textit{lnode}.\Omega \text{ do}
12: \hspace{3em} \textbf{if} \ \gamma(\pi_{d_k}(\Omega_i)) \cap \pi_k(\textit{cdir}.B^L_k) = \gamma(\pi_{d_k}(\Omega)) \text{ then}
13: \hspace{4em} \text{UpdateBucketInfo(\Omega_i, \textit{cdir}.B^L_k)}
14: \hspace{2em} \text{Partitioning(\textit{cdir}.B^L_k.\textit{cnode})}
15: \hspace{2em} \text{Clustering(\textit{cdir}.B^L_k)}
16: \hspace{2em} \text{Clustering(\textit{cdir}.B^R_k)}
17: \hspace{2em} \text{UpdateClusterCapacity(\textit{lnode})}
18: \hspace{2em} \textbf{else if} \ \gamma(\pi_{d_k}(\Omega_i)) \cap \pi_k(\textit{cdir}.B^R_k) = \gamma(\pi_{d_k}(\Omega)) \text{ then}
19: \hspace{3em} \text{UpdateBucketInfo(\Omega_i, \textit{cdir}.B^R_k)}
20: \hspace{2em} \text{Partitioning(\textit{cdir}.B^R_k.\textit{cnode})}
21: \hspace{2em} \text{Clustering(\textit{cdir}.B^R_k)}
22: \hspace{2em} \text{UpdateClusterCapacity(\textit{lnode})}

\[
\text{buckets}_{\text{opt}} = \begin{cases} 
\text{split}_{\text{dataClustering}}(\text{find}_{\text{opt}}(\Omega_{\text{low}}, \Omega_{\text{up}})) & \text{if } |B^O_k| \geq \alpha \land |B^L_k| \geq \alpha \\
\text{split}_{\text{spaceClustering}}(\Omega, s) & \text{otherwise,}
\end{cases}
\]

where \( s = \chi_k(B^k) \), \( B^k \) is the overflowing bucket at the \( k^{th} \) level of \( BE^* \)-Tree, and \( B^O_k, B^L_k, B^R_k \) are the new right, overlapping, and left buckets, respectively, returned by either \text{split} functions. Furthermore, the most cost effective splitting, denoted by \text{find}_{\text{opt}} \), is determined by a set of buckets (right, overlapping, left) which minimizes the total overlap, to break ties among candidate bucket sets, the set with the smallest coverage is selected. The function \text{find}_{\text{opt}} \) is formally defined as follows:
\[ (\Omega_j, I) \leftarrow \text{find}^{\text{opt}} (\Omega_{\text{low}}, \Omega_{\text{up}}) \leftarrow \arg \min_{(\Omega_j, I) \in \text{minoverlap}(\Omega_{\text{low}}, \Omega_{\text{up}})} \text{coverage}(\Omega_j, I) \] (5.16)

\[
\{(\Omega_j, I)\} \leftarrow \text{minoverlap}(\Omega_{\text{low}}, \Omega_{\text{up}}) \leftarrow \left\{ \arg \min_{\Omega_j \in \{\Omega_{\text{low}}, \Omega_{\text{up}}\}, \ I \in (\alpha \cdots |\Omega| - \alpha)} \text{overlap}(\Omega_j, I) \right\},
\]

(5.17)

where \(\Omega_{\text{low}}\) represents the expressions in the overflowing bucket \(B^k\) sorted in descending order w.r.t. the lower bound of the expressions’ \(k^{\text{th}}\) dimension; similarly, \(\Omega_{\text{up}}\) represents the expressions sorted in ascending order w.r.t. the upper bound on the \(k^{\text{th}}\) dimension.

The overlap and coverage are computed over \(k^{\text{th}}\) dimension of the bucket as follows:

\[
\text{overlap}(\Omega, I) = \left( \lambda_k \left( \pi_k(B^k_L) \right) \mid B^k_L, B^k_O, B^k_R \leftarrow \text{split}^{\text{opt}} \text{dataClustering}(\Omega, I) \right)
\]

(5.18)

\[
\text{coverage}(\Omega, I) = \left( \lambda_k \left( \pi_k(B^k_L) \cup \pi_k(B^k_O) \cup \pi_k(B^k_R) \right) \mid B^k_L, B^k_O, B^k_R \leftarrow \text{split}^{\text{opt}} \text{dataClustering}(\Omega, I) \right).
\]

(5.19)

The descendant-repelling and overlap-free buckets are defined as follows,\(^5\) which is the basis of our data clustering procedure:

\[
(B^k_L, B^k_O, B^k_R) \leftarrow \text{split}^{\text{opt}} \text{dataClustering}(\Omega, I) =
\]

\[
\bigcup_{i=1}^{\left|\Omega\right|} \left( B^k_L \leftarrow \Gamma_{\min}^{\pi_{d_{i-1}} \cdots d_k} (\Omega_i) \right),
\]

\[
\bigcup_{i=I+1}^{\left|\Omega\right|} \left( (B^k_R \leftarrow \Gamma_{\min}^{\pi_{d_{i-1}} \cdots d_k} (\Omega_i) \mid \pi_k(B^k_L) \cap \gamma(\pi_{d_k}(\Omega_i)) = \emptyset) \right),
\]

\[
(B^k_O \leftarrow \Gamma_{\min}^{\pi_{d_{I-1}} \cdots d_k} (\Omega_i) \mid \pi_k(B^k_L) \cap \gamma(\pi_{d_k}(\Omega_i)) \neq \emptyset) \right).
\]

(5.20)

In contrast, if no data clustering exists that produces left and right buckets with at least \(\alpha\) expressions, then we rely on space clustering, a default clustering which divides the space in half in order to avoid degeneration of the clustering directory.

\(^5\)The provided definition of \(\text{split}^{\text{dataClustering}}\) is used when computing buckets over \(\Omega_{\text{up}}\); for applying it to \(\Omega_{\text{low}}\), the computation of \(B^k_L\) and \(B^k_R\) must simply be reversed.
Our ranking function simplifies BE-Tree’s ranking function while maintaining the actual index objective, namely, reducing the matching cost. The matching cost is defined as the total number of predicate evaluations: \textit{minimizing false candidate computations} and \textit{minimizing true candidate computations}. The former cost is the total number of predicate evaluations for each discarded search path or for each unsatisfied expression, i.e., penalizing multiple search paths or paths resulting in many false positives. The latter cost is the number of predicate evaluations before concluding a set of expressions as matched, i.e., promoting evaluation of common predicates among expressions exactly once.

Furthermore, our ranking model relies on the notion of \textit{false candidates}: the scanned expressions (subscriptions) that are not matched by an input expression (events). Our model assigns a rank to each node \( n_i \) using the function \( \text{rank}(n_i) \) which is a combination of the \( \text{loss}(n_i) \) and \( \text{gain}(l_j) \) functions. \( \text{loss}(n_i) \) computes for each node the false candidates generated over a window of \( m \) events. \( \text{gain}(l_j) \) computes the number of common predicates for each of its expressions. Formally, \( \text{rank}(n_i), \text{loss}(n_i), \) and \( \text{gain}(l_j) \) are defined as follows:

\[
\text{loss}(n_i) = \sum_{i} \mu(\pi_{\text{db}}(\Omega_i)) > s, \\
\text{gain}(l_j) = \sum_{i} \pi_{\text{db}}(\Omega_i) \cap \pi_k(B^k_l) = \emptyset.
\]

### 5.4.2 Adaptive Ranking Function

Approximate Splitting Algorithm Unlike the exhaustive algorithm, the cost of our fast splitting algorithm is dominated by sorting expressions, namely, \( O(n \log(n)) \), where \( n = |\Omega| \). In a nutshell, the approximate algorithm assigns expressions one-by-one from the leftmost expression to the left bucket and from the rightmost expression to the right bucket while pushing expressions that span both left and right bucket into the overlap bucket. Formally the algorithm is defined as follows:

\[
\text{buckets}_{\text{apx}} = \begin{cases} 
\text{split}_{\text{dataClustering}}(\Omega_{1\text{v}}, \Omega_{\text{up}}) & \text{if } |B^k_R| \geq s, \\
\text{split}_{\text{dataClustering}}(\Omega, s) & \text{otherwise}, 
\end{cases}
\]

where \( s = \chi_k(B^k) \).

\[
(B^k_L, B^k_O, B^k_R) \leftarrow \text{split}_{\text{dataClustering}}(\Omega, s) = \begin{cases} 
\bigcup_{i=1}^{\Omega_{\text{up}}} (B^k_L \leftarrow \Gamma_{\text{min}}^{\pi_{\text{db}}(\Omega_{\text{up}})}(\Omega_{\text{up}}) \cap \pi_k(B^k_R) = \emptyset), \\
(B^k_R \leftarrow \Gamma_{\text{min}}^{\pi_{\text{db}}(\Omega_{\text{up}})}(\Omega_{\text{up}}) \cap \pi_k(B^k_L) = \emptyset), \\
\bigcup_{i=1}^{\Omega_{\text{up}}} (B^k_O \leftarrow \Gamma_{\text{min}}^{\pi_{\text{db}}(\Omega_{\text{up}})}(\Omega_{\text{up}}) \not\in B^k_L \land \Omega_{\text{up}} \not\in B^k_R).
\end{cases}
\]
The proposed ranking model is simply generalized for splitting an overflowing node \( l_j \)-node using a new \( \text{attr}_i \). It is given by

\[
\text{rank}(l_i) = \text{gain}(l_i) - \text{loss}(l_i),
\]

where \( \text{gain}(l_i) \) is approximated by the number of expressions that have a predicate on \( \text{attr}_i \), and \( \text{loss}(l_i) \) is estimated either using the selectivity by constructing a histogram, in which \( \text{loss}(l_i) \) is the average bucket size in the histogram, or using popularity, an optimistic approach, by initially setting \( \text{loss}(l_i) \) to zero to eliminate any histogram construction and to rely on the matching feedback mechanism for adjusting the ranking if necessary. In our experimental evaluation, the optimistic approach results in an improved matching and insertion time.

### 5.5 Top-k Algorithm

Our \( \text{top-k} \) algorithm is a two-stage process in which, on the one hand, a cost-based tree traversal is utilized to determine which \( p \)-node in \( \text{BE}^* \)-Tree to visit next, and, on the other hand, along the traversal
Figure 5.4. for storing upper bound scores within each l-node. This compression results in huge space reduction. Moreover, the same compression technique is applied and the root’s relevant p-i-nodes are computed to form a key (score_{p_i}, p_i-node) which only the information about the attribute space (and the upper bound score for each attribute) is maintained, no maintenance of scoring w.r.t. the distribution of values within each dimension is required. As a result, the upper bound scoring information that must be stored at each p-node can be compressed. This compression results in huge space reduction. Moreover, the same compression technique is applied for storing upper bound scores within each l-node. The overall BE*-Tree top-k structure is illustrated in Figure 5.4.

The top-k algorithm is as follows: Upon arrival of an event expression, the score between the event and the root’s relevant p_i-nodes are computed to form a key (score_{p_i}, p_i-node) which is then inserted into the p-node-max-heap (Algorithm 14 Lines 6-12). Next, the p-node with highest score value is removed from the p-node-max-heap and its subtree is visited (Algorithm 14 Lines 13-19). For the chosen p-node, all of its relevant c_j-nodes (those c_j-nodes having a non-empty intersection with the event) are visited while going through the c-directory (Algorithms 15). For each c_j-node, its l-node is scanned. This

---

Algorithm 14 MatchTopk(\(\omega, \Omega_{matched}, cnode\))

1: InodeScore = uscore(\(\omega, lnode, sum_{\Omega}^l\));
2: if InodeScore < MinScore(k-min-heap) then
3: /* Inode pruning, skipping an l-node*/
4: else
5: \(\Omega_{matched} \leftarrow\) FindMatchingExpr(cnode,lnode)
6: /* Iterate over event’s predicates*/
7: for \(P_{attr} \in \omega\) do
8: /* Retrieve c-node’s p-directory for \(P_{attr}\) */
9: pnode \(\leftarrow\) SearchPDir(\(P_{attr}, cnode.pdir\))
10: /* If \(P_{attr}\) exists in the p-directory*/
11: if pnode \(!=\) NULL then
12: pnodeScore = uscore(\(\omega, pnode, sum_{\Omega}^p\));
13: if pnodeScore > MinScore(k-min-heap) then
14: p-node-max-heap \(\leftarrow\) (pnode, pnodeScore)
15: while !IsEmpty(k-min-heap) do
16: maxScorePnode \(\leftarrow\) GetMax(p-node-max-heap)
17: if IsFull(k-min-heap) then
18: break /* pnode pruning, matching termination*/
19: SearchCDir(\(\omega, \Omega_{matched}, maxScorePnode, cd\))
20: DeleteMax(p-node-max-heap)

Algorithm 15 SearchCDir(\(\omega, \Omega_{matched}, cd\))

1: if \(\gamma(p_d(\omega)) \cap \pi_k(cd.B_{R}^k) \neq \emptyset\) then
2: MatchTopk(\(\omega, \Omega_{matched}, cd.B_{R}^k, cnode\))
3: if \(\gamma(p_d(\omega)) \cap \pi_k(cd.B_{L}^k) \neq \emptyset\) then
4: SearchCDir(\(\omega, \Omega_{matched}, cd.B_{L}^k\))
5: if \(\gamma(p_d(\omega)) \cap \pi_k(cd.B_{R}^k) \neq \emptyset\) then
6: SearchCDir(\(\omega, \Omega_{matched}, cd.B_{R}^k\))
process is applied recursively until either the top-k expressions are discovered or all the relevant portions of BE*-Tree are visited. We now discuss the pruning procedures (top-k early terminations):

1. **l-node pruning** Before scanning expressions in an l-node, given an event expression, the upper bound score of an l-node is computed, if the score is larger than the minimum score of the matched expressions seen so far, stored in k-min-heap, or there are fewer than k matched expressions seen so far, then the l-node is visited; it is pruned otherwise, as outlined in Algorithm 14, Line 2.

2. **p-node pruning** Before inserting the p-node into the p-node-max-heap, the score of p-node w.r.t. the event expression is computed; only if the score is larger than the minimum score of matched expressions (or fewer than k matched expressions are gathered so far), then the p-node is inserted. Similarly, before removing the p-node with the highest score from the p-node-max-heap, the score of the highest p-node is compared with the lowest score of matched expressions, if the p-node’s score is smaller, then the entire matching process is terminated, resulting in a substantial saving in the matching time, as given in Algorithm 14, Lines 15-17 results.

### 5.6 Evaluations

We present an extensive evaluation of BE*-Tree based on synthetic and real datasets. The experiments were ran on a machine with two Intel Xeon 3.00GHz Quad-core processors having 16GB of memory and running CentOS 5.5 64bit. All algorithms are implemented in C and compiled using gcc 4.1.2.

#### 5.6.1 Experiment Overview

Our evaluation considers the following algorithms: (1) SCAN (a sequential scan), (2) k-ind (the conjunction algorithm implemented over k-index [211] in which all subscriptions are known in advance), (3) BE (BE-Tree [176]), (4) BE*-B (a batch-oriented construction of BE*-Tree in which all subscriptions are known in advance resulting in better initial statistics), (5) BE* (a fully dynamic version of BE*-Tree in which subscriptions are inserted individually). Most notably, BE*-Tree is the only structure that is designed to support continuous domains (with the exception of the naive SCAN). Therefore, to compare BE*-Tree with state-of-the-art algorithms, we emulate the effect of continuous domains using a large discrete domain. Furthermore, we primarily focus on the range operator (BETWEEN) over discrete domains. Moreover, we further restrict the size of ranges to only a handful of values, otherwise the enumeration technique of k-index [211] to support range operators would become infeasible due to exponential space explosion.

Our evaluation explores various crucial workload characteristics including workload distribution, workload size, space dimensionality, average subscription and event size, dimension cardinality, number of clusters in each dimension, dimension cluster size, and event matching probability. To generate various experimental configurations, we leverage the BEGen framework [176] including public domain data (i.e., DBLP repository data) to generate real workloads. Various workload configurations are captured in Tables 5.2-5.3. In this section, we primarily focus on a distinguishing subset of experimental results, proven to be interesting, extracted from Table 5.2.

---

### Table 5.2: Synthetic Workload Configuration

<table>
<thead>
<tr>
<th>Size</th>
<th>Number of Dimensions</th>
<th>Dimension Cardinality</th>
<th>Number of Clusters</th>
<th>Cluster Size</th>
<th>Sub/Event Size</th>
<th>Match Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K-1M</td>
<td>400</td>
<td>250M</td>
<td>4</td>
<td>100K</td>
<td>100K</td>
<td>1M</td>
</tr>
<tr>
<td>50-1400</td>
<td>400</td>
<td>250M</td>
<td>4</td>
<td>100K</td>
<td>100K</td>
<td>1M</td>
</tr>
<tr>
<td>2.5K-250M</td>
<td>400</td>
<td>250M</td>
<td>4</td>
<td>100K</td>
<td>100K</td>
<td>1M</td>
</tr>
<tr>
<td>250M</td>
<td>400</td>
<td>250M</td>
<td>4</td>
<td>100K</td>
<td>100K</td>
<td>1M</td>
</tr>
<tr>
<td>250M</td>
<td>400</td>
<td>250M</td>
<td>4</td>
<td>100K</td>
<td>100K</td>
<td>1M</td>
</tr>
<tr>
<td>250M</td>
<td>400</td>
<td>250M</td>
<td>4</td>
<td>100K</td>
<td>100K</td>
<td>1M</td>
</tr>
<tr>
<td>250M</td>
<td>400</td>
<td>250M</td>
<td>4</td>
<td>100K</td>
<td>100K</td>
<td>1M</td>
</tr>
</tbody>
</table>

### Table 5.3: Real Workload Configuration

<table>
<thead>
<tr>
<th>Size</th>
<th>DBLP (Author)</th>
<th>DBLP (Title)</th>
<th>Match Prob (Author)</th>
<th>Match Prob (Title)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100-760K</td>
<td>267</td>
<td>267</td>
<td>0.01-9</td>
<td>0.01-9</td>
</tr>
<tr>
<td>50-250K</td>
<td>267</td>
<td>267</td>
<td>0.01-9</td>
<td>0.01-9</td>
</tr>
<tr>
<td>400k</td>
<td>267</td>
<td>267</td>
<td>0.01-9</td>
<td>0.01-9</td>
</tr>
<tr>
<td>150</td>
<td>267</td>
<td>267</td>
<td>0.01-9</td>
<td>0.01-9</td>
</tr>
</tbody>
</table>

### Table 5.4: BE*-Tree Micro Experiments

#### (a) Max Leaf Capacity Trends

<table>
<thead>
<tr>
<th>A</th>
<th>Data Sets</th>
<th>Time (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Unif (1M)</td>
<td>2.30</td>
</tr>
<tr>
<td>20</td>
<td>Zipf (1M)</td>
<td>2.45</td>
</tr>
<tr>
<td>100</td>
<td>Author (760K)</td>
<td>4.98</td>
</tr>
<tr>
<td>43</td>
<td>Title (250K)</td>
<td>2.83</td>
</tr>
</tbody>
</table>

#### (b) Construction Time

<table>
<thead>
<tr>
<th>A</th>
<th>Data Sets</th>
<th>Time (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Unif (1M)</td>
<td>2.30</td>
</tr>
<tr>
<td>20</td>
<td>Zipf (1M)</td>
<td>2.45</td>
</tr>
<tr>
<td>100</td>
<td>Author (760K)</td>
<td>4.98</td>
</tr>
<tr>
<td>43</td>
<td>Title (250K)</td>
<td>2.83</td>
</tr>
</tbody>
</table>

### 5.6.2 Micro Experiments

#### BE*-Tree Parameters

BE*-Tree internal parameters are extensively studied and the most robust configuration that resulted in the best overall performance in all experiments is presented here (the in-depth results are omitted). The values used throughout our experiments are $A$ (maximum leaf capacity) shown in Table 5.4(a); $\alpha = 10\%$ (minimum capacity of descendant buckets); $\beta = 4$ (maximum allowable bucket boundary increase before invoking bottom-up expansion); and the exhaustive splitting algorithm is used to handle overflowing buckets.

#### BE*-Tree Construction

The dynamic construction time of BE*-Tree for our largest workload having up to a million subscriptions was below 5 seconds; BE*-Tree’s average construction time over our main datasets are reported in Table 5.4(b).
5.6.3 Macro Experiments

In our macro experiments, we first establish the effectiveness of BE*-Tree’s internal techniques, namely, the non-rigid space-cutting, bi-directional tree expansion, and descendant-repelling and overlap-free splitting, then, we shift our focus to BE*-Tree’s hierarchical top-k processing that is realized over these unique inner structures of BE*-Tree for finding the most relevant matches.

Workload Distribution

A key distinguishing property that sets apart key- vs. non-key-based and rigid-based vs. non-rigid-based algorithms is the workload distribution. The effects of the distributions are shown in Figure 5.5-5.12 in which the graphs in the left column correspond to the uniform distribution (for choosing predicates’ attributes) while the graphs on the right column correspond to the Zipf distribution with the exception of DBLP-driven workloads which follow a real-data distribution. The observed trend is that k-index, a non-key-based method, is highly sensitive to the underlying data distribution because few popular attributes appear in the majority of subscriptions which result in increased false candidates. The presence of popular attributes forces k-index to sort and scan through posting lists with large number of subscriptions [211]. The increased false candidate problem is reduced by BE-Tree, being a key-based approach. However, BE-Tree also needlessly indexes dead spaces. As a result, BE*-Tree on average outperforms k-index by 60% and 90% and BE-Tree by 35% and 65% for the uniform and the Zipf distributions, respectively.
Chapter 5. Top-k Query Indexing (Main Memory)

114

Figure 5.6: Varying Space Dimensionality

Workload Size

As subscription workload size increases all algorithms scale linearly w.r.t. the number of matched subscriptions. Yet BE*-Tree achieves an 37% improvement compared to the next best algorithm for the uniform workload and a 68% better matching time for the Zipf workload, as shown in Figure 5.5a,b. Similar trends are also observed for real datasets, in which for the author dataset (with an average of 8 predicates per subscription), improvements of 38% over BE-Tree and 94% over k-index is achieved (Figure 5.5c). For the title dataset having larger number of predicates per subscription (with an average of 35 predicates), the gap between BE*-Tree and the other algorithms further widens to over 59% for BE-Tree and 95% for k-index (Figure 5.5d). This is due to the novel scoring that exploits interrelationships within dimensions and the non-rigid multi-layer structure of BE*-Tree to effectively reduce the search space.

Dimensionality

Space dimensionality is another factor that distinguishes between the key and non-key-based methods; consequently, k-index, a non-key-based approach, substantially suffers in lower dimensionality for the uniform workload due to higher degree of overlap among subscriptions. The increased overlap results in increased false candidates. As the dimensionality decreases from 1400 to 50 k-index, the matching time also increases by a factor of 6.14x, see Figure 5.6a. However, for the Zipf distribution, there is little change in matching time as dimensionality increases because the few popular dimensions under the Zipf distribution continue to generate many false candidates (see Figure 5.6b). Also varying the dimensionality does not significantly effect other approaches, and BE*-Tree remains to be the top performing algorithm.

Subscription/Event Size

A subtle workload characteristic is the average number of predicates per subscription and event. The most effected algorithm is k-index because an increase in the number of subscription predicates translates into an increase in the overlap among subscriptions, and an increase in the number of event predicates translates into retrieval of a larger number of k-index’s posting lists. In general, BE*-Tree gracefully scales as the number of predicates increases because of its non-rigid multi-layer structure and improves over the next best algorithm by 39% for the uniform and by 66% for the Zipf workload, as illustrated by
An important factor that further differentiates rigid from non-rigid algorithms is dimension cardinality. For a rigid algorithm, BE-Tree, the larger cardinality implies larger tree depth because BE-Tree relies on a grid-based approach that recursively cuts space in half. Therefore, as the cardinality increases, BE-Tree’s matching time increases by a factor of 1.29x and 1.36x for the uniform and the Zipf distributions, respectively, while BE*-Tree remains almost unaffected as the cardinality increases and continues to improve over the next best approach by up to 65% (see Figure 5.8a,b).

Another consequence of increasing the dimension cardinality is the effect on memory footprint. Especially for k-index, increasing dimension cardinality results in an exponential memory growth when the ratio of predicate range size to dimension cardinality is kept constant. The space blow-up occurs because k-index relies on the enumeration technique to support range predicates. For instance, in order to cope with the operator BETWEEN \([v_1, v_2]\), the enumeration essentially transforms the value of \(v_2 - v_1\) from a decimal to a unary representation: an exponential transformation.

Number of Clusters/Cluster Size

Next, we study the effects of the number of clusters and the cluster size in each domain. If no clustering is employed, then each predicate draws its value randomly from its domain, but when clustering is
employed, the drawn values are clustered in certain regions. For example, when the number of clusters is 10, then all drawn values are centered (clustered) at 10 randomly chosen points in the domain. On the other hand, the cluster size models the extent of each individual cluster. For instance, the cluster size 400 implies that all values belonging to a particular cluster are within the Euclidean distance of 400. Studying these workload characteristics are essential when the data is being generated syntactically over a large domain.

k-index is highly sensitive to both the number of clusters and the cluster size. Both fewer clusters and smaller cluster size results in larger overlap among subscriptions and increases the number of false candidates, especially for the Zipf distribution. Thus, from having no cluster, denoted by “N/A”, to only two clusters, the matching time is increased by a factor of 20.84x, and from reducing the cluster size from 800 to 25, the matching time is also increased by a factor of 17.44x (Figure 5.9b,d).

Likewise, BE-Tree, by relying on a rigid clustering, is best suited when predicate values are uniformly drawn, namely, no data clustering, because the rigid clustering fails to fully adapt to the skewed distribution and fails to avoid indexing the dead spaces. As a result, BE*-Tree improves over BE-Tree by up to 77% when data is clustered and by up to 65% when no clustering is used (Figure 5.9b). Finally, BE*-Tree remains dominant as the number and size of clusters are varied, as observed in Figure 5.9.

### Percentage of Matching Probability

In general, an increase in matching probability results in an increase in the number of candidate matches. Therefore, k-index is forced to scan large posting lists with a reduced chance of pruning and an increased application of sorting to advance through posting lists. For the DBLP title dataset increased matching is
Chapter 5. Top-k Query Indexing (Main Memory)

<table>
<thead>
<tr>
<th>Match Prob</th>
<th>Sub=1M</th>
<th>Sub=400K</th>
<th>Sub=150K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zipf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Title</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Further signified due to larger subscription and event size which further increases the number of posting lists that must be visited (Figure 5.10). The increased matching time is also observed in both BE-Tree and BE*-Tree simply due to an increased number of matches. However, these latter techniques scale better as matching time increases; in particular, BE*-Tree continues to outperform the next best algorithm even as the matching probability reaches 9% by 39%, 39%, 21%, and 16% for the uniform, Zipf, author, and title datasets, respectively, as shown in Figure 5.10.

\[ \forall P_j(x) \in \Omega_i, P^*_j = \max(\text{inv}(P^\text{attr}_j), N(0.8 \times \text{inv}(P^\text{attr}_j), 0.05 \times \text{inv}(P^\text{attr}_j))) \]  
(5.27)

**Workload Size/Matching Probability (top-k)**

To enable top-k processing, we must first generate the predicate weights. In fact, our model is not limited to any particular weight generation technique or scoring function for top-k. Therefore, we adopt the weight generation technique proposed in [211]. In brief, for each unique attribute, we first compute its inverse frequency, denoted by inv(attr), i.e., a popular attribute will be assigned a low weight while a rare attribute will be assigned a high weight. Second, for each expression \( \Omega_i \), we compute its predicate weight using a random Gaussian distribution (adopted from [211]) as follows:

Next, we present BE*-Tree’s top-k processing results for varying the workload size (Figure 5.11) and the matching probability (Figure 5.12). BE*-Tree hierarchical top-k processing exhibits a significant performance gain compared to k-index for finding the top-1 (denoted by \( k-\text{ind}(1) \) and BE*(1)) and top-5
(BE*(5)) most relevant expression(s). BE*-Tree’s success is attributed to its hierarchical structure that refines the upper bound score as traversing down the tree; however, k-index is restricted to a flat structure that is unable to further refine the upper bound score as more expressions are seen and limited to a fix top-layer upper bound scoring. In short, as we increase the workload size (Figure 5.11), BE*(1) improves over BE*-Tree by up to 23%, 77%, 72%, 71% while k-ind(1) improves over k-index by only up to 1.3%, 6.2%, 3.5%, 8.5%\(^7\) for the uniform, Zipf, author, and title datasets, respectively. Similar trends that show BE*-Tree’s effectiveness are also observed as we increase the matching probability (Figure 5.12). In general, an increase in matching probability results in an increase in the number of candidate matches. Therefore, k-index is forced to scan large posting lists with a reduced chance of pruning and an increased application of sorting to advance through posting lists. The increased matching time is also observed in BE*-Tree simply due to an increased number of matches. However, BE*-Tree scales better as matching time increases and continues to outperform k-ind.

5.7 Summary

To address the problem of finding the most relevant matches, we present BE*-Tree, a data structure that employs an effective hierarchical top-k pattern matching algorithm. Furthermore, BE*-Tree introduces a novel non-rigid space-cutting technique to index Boolean expressions over a high-dimensional space by developing (1) a bi-directional tree expansion technique that enables indexing only non-empty continuous

\(^7\)Similar minor improvements were also reported in [211] that showed k-index’s top-k achieved only up to 11% improvement over the baseline on some datasets while resulted in up to 19% performance loss on others.
sub-spaces and (2) a descendant-repelling and overlap-free splitting strategy.

Moreover, BE*-Tree is a general index structure for efficiently processing large collections of Boolean expressions, a core functionality, required by a wide range of applications including (complex) event processing, algorithmic trading, targeted advertising, selective information dissemination, and location-based services. Finally, the performance of BE*-Tree is proven through a comprehensive experimental comparison with state-of-the-art index structures for matching Boolean expressions.

Figure 5.12: Top-k vs. Matching Probability
Chapter 6

Data Indexing (SSD)

6.1 Introduction

In a multiversion database system, new records do not physically replace old ones. Instead, a new version of the record is created, which becomes visible to other transactions at commit time. Conceptually, there may be many rows for a record, each corresponding to the state of the database at some point in the past. Very old versions may be garbage-collected as the need for old data diminishes, in order to reclaim space for new data.

When indexing data, one typically indexes only the most recent version of the data, since that version is most commonly accessed. In such a setting, record insertions, deletions and updates trigger I/O to keep the indexes up to date. With a traditional index structure, the deletion of a record requires the traversal of each index and the removal of the row-identifier (RID) from the leaf node. The update of a record (changing one attribute value to another) creates a new version, triggering a traversal of all indexes to change the RIDs to the new version’s RID. (In the case of the modified attribute, the position of the record in the index may also change). For a newly inserted record, the new RID must be inserted into each index. Indexes may be large, and in aggregate much too large to fit in the RAM bufferpool. As a result, all of these index maintenance operations will incur the overhead of physical I/O on the storage device.

These overheads have historically been problematic for OLTP workloads that are update-intensive. As a result, OLTP workloads are often tuned to minimize the number of indexes available. This choice makes it more difficult to efficiently process queries and to locate records based on secondary attributes. These capabilities are often important for operational data stores.

Our goal is to reduce the overhead of index updates, so that indexes can be used effectively for analytical query processing without being a heavy burden on transaction throughput. The query vs. update dilemma is clearly captured in Figure 6.1, a preview of our experimental results. The execution time for analytical queries is reduced as more indexes are added. However, this reduction comes at the cost of increasing the update time in the Base approach. In contrast, by employing our technique (the Indirection approach) the incurred update cost is significantly smaller.

To address this dilemma, we utilize a solid state storage layer. Based on current technologies, solid state disks (SSDs) are orders of magnitude faster than magnetic disks (HDDs) for small random I/Os. However, per gigabyte, SSDs are more expensive than magnetic disks. It therefore pays to store the bulk
of the data on magnetic disk, and reserve the SSD storage for portions of the data that can benefit the most, typically items that are accessed frequently and randomly.

Unlike previous approaches [40, 121, 31, 41, 68], we do not propose to simply store “hot” data on SSDs. Instead, we change the data structures in the storage hierarchy to employ an extra level of indirection through solid state storage. Because the solid state memory is fast, the extra time incurred during index traversal is small, as we demonstrate experimentally. The extra level of indirection dramatically reduces the amount of magnetic disk I/O that is needed for index updates. Only SSD I/O is needed for deletions and updates, with the exception of indexes on changed attributes. We can also reduce the magnetic disk I/O overhead for insertions. While we describe our techniques in terms of SSDs, we are not limited to a disk-like form factor. In fact, alternative form factors (e.g., FusionIO auto-commit memory [86]) with smaller I/O granularities would provide even better performance because our proposed solid state updates are small.

### Multiversion Databases

By keeping old data versions a system can enable queries about the state of the database at points in the past. A simple implementation of a multiversion database would store the row-identifier (RID) of the old version within the row of the new version, defining a linked list of versions. Such an implementation allows for the easy identification of old versions of each row, but puts the burden of reconstructing consistent states at particular times on the application, which would need to keep timing information within each row.

To relieve applications of such burdens, a multiversion database system can maintain explicit timing information for each row. In a valid time temporal model [88] each row is associated with an interval [begin-time, end-time) for which it was/is current. Several implementation choices exist for such a model. One could store the begin-time with each new row, and infer the end-time as the begin-time of the next version. Compared with storing both the begin-time and end-time explicitly for each row, this choice saves space and also saves some write I/O to update the old version. On the other hand, queries over historical versions are more complex because they need to consult more rows to reconstruct validity.
A *bitemporal* database maintains two kinds of temporal information, the system (i.e., transaction) time, as well as the application time (sometimes called “business time”).

In this chapter, we do not commit to any one of these implementation options, each of which might be a valid choice for some workloads. For any of these choices, our proposed methods will reduce the I/O burden of index updates. Some of our techniques, such as the LIDBlock technique (Section 6.3.1), apply to both versioned and non-versioned databases.

**Physical Organization**

There are several options for the physical organization of a temporal database. A complete discussion of the alternatives is beyond the scope of this chapter. We highlight two options that have been used in commercial systems.

One organization option appends old versions of records to a “history” table and only keeps the most recent version in the main table, updating it in-place. Commercial systems have implemented this technique: In IBM DB2 it is called “System-period data versioning” [110], and it is used whenever a table employs transaction time as the temporal attribute. The Oracle Flashback Archive [163] also uses a history table. Such an organization clusters the history table by end-time, and does not impose a clustering order on the main table. Updates need to read and write the main table, and also write to the end of the history table. Because updates to the main table are in-place,¹ an index needs to be updated only when the corresponding attribute value changes. For insertions and deletions, all indexes need to be updated.

In this chapter, we assume an organization in which there is a single table containing both current and historical data. Commercial systems that implement this technique include Oracle 11g where the concept is called “version-enabled tables” [162]. IBM’s DB2 also uses this approach for tables whose only temporal attribute is the application time. New rows are appended to the table, so that the entire table is clustered by begin-time. Updates need to read the table once and write a new version of the record to the end of the table.

We focus on applications that primarily use current data, but occasionally need to access older versions of the data. To support queries over current data, the most recent data may be extensively indexed. Older data may be less heavily indexed because it is queried less frequently, and is often more voluminous. Even within a single table, the system can offer an implementation option in which only the most recent version of a record appears in an index.

### 6.2 Basic Indirection Structure

Traditional index structures directly reference a record via a pointer known as a physical row-identifier (RID). The RID usually encodes a combination of the database partition identifier, the page number within the partition, and the row number within the page. A RID index over current HDD-resident data is shown in Figure 6.2.

¹If one wanted to cluster the main table by a temporal attribute to improve temporal locality, then updates would not be in-place and additional indexes would need to be updated. Our proposed solution would reduce the burden of such index updates.
The choice of a physical identifier hinders the update performance of a multiversion database in which updates result in a new physical location for the updated record. Changes to the record induce I/O for every index, even indexes on “unaffected” attributes, i.e., attributes that have not changed. Random I/Os are required to modify HDD-resident leaf pages.

To avoid HDD I/O for indexes on unaffected attributes, we decouple the physical and logical representations of records spanning many versions. We distinguish between a physical row-identifier (RID) and a logical record identifier (LID). For any given record, there may be many RIDs for that record corresponding to the physical placement of all of the versions of that record. In contrast, the LID is a reference to the RID representing the most recent version of the record. For now, one can think of a table $LtoR(LID, RID)$ that has LID as the primary key. Indexes now contain LIDs rather than RIDs in their leaves.

Under our proposed Indirection technique, an index traversal must convert a LID to a RID using the $LtoR$ table. A missing LID, or a LID with a NULL RID in the $LtoR$ table are treated as deleted rows, and are ignored during search. By placing the $LtoR$ table on an SSD, we ensure that the I/O overhead for the extra indirection is relatively small. Because the SSD is persistent, index structures can be recovered after a crash. Because we need only a few SSD bytes per record, it is possible to handle a large magnetic disk footprint with a much smaller solid state footprint. The new index design is demonstrated in Figure 6.3.

When an existing record is modified, a new version of that record is created. The $LtoR$ table is updated to associate the new row’s RID to the existing LID. That way, indexes on unchanged attributes remain valid. Only for the changed attribute value will index I/O be required for the indirection layer.

When a record is deleted, the (LID,RID) pair for this record in the $LtoR$ table is deleted. Index traversals ignore missing LIDs. Indexes can lazily update their leaves during traversal, when a read I/O is performed anyway. At that time, any missing LIDs encountered lead to the removal of those LIDs from the index leaf page. After a long period of activity, indexes should be validated off-line against the

\[\text{Figure 6.2: Traditional RID index structure}\]
When a new record is added, the new record is appended to the tail of the relation and its RID is fetched and associated to a new LID. The (LID, RID) pair for the new record is added \textit{LtoR} table. All indexes are also updated with the new record LID accordingly. In Section 6.3, we discuss how to further improve record insertion and deletion.

6.3 Enhancing Insertions

We now develop techniques for improving the index performance of insertion. We define a batching structure called a \textit{LIDBlock}, and employ yet another level of indirection.

6.3.1 LIDBlocks

To reduce the index overhead for insertions, we propose an SSD-resident auxiliary \textit{LIDBlock} structure containing a fixed number of LIDs. The LIDs in a \textit{LIDBlock} may be NULL, or may valid LIDs from the \textit{LtoR} table. References to a \textit{LIDBlock} are mapped to multiple LIDs through this extra level of indirection. Figure 6.4 shows the extended structure with \textit{LIDBlocks}.

The arrow from index leaf pages to \textit{LIDBlocks} in Figure 6.4 could be implemented by keeping a block identifier (BID) within the leaf page. The disadvantage of such a choice is that the leaf node of the index needs to be read from magnetic disk to locate the BID, requiring extra HDD I/O. Instead, we propose to store \textit{LIDBlocks} within hash tables on the SSD. In the following sections, we describe more precisely
how index pages refer to and traverse LIDBlocks.

6.3.2 The Dense Index Case

Consider first the case of a dense index, i.e., a secondary index where there are many records (more than a leaf-node’s worth) for each attribute value. For such indexes, we keep a list of LIDs for each value as before. In addition, we store a collection of LIDBlocks on the SSD in a hash table, hashed by the attribute value. Initially, each indexed value has a LIDBlock whose LIDs are all NULL.

When a new record is inserted, we need to modify the indexes to reflect the insertion. Suppose that the value of an indexed attribute is \( v \), and that the index is dense. A LID is created for the record, and a suitable (LID,RID) pair is added to the LtoR table. The LIDBlock \( B \) for \( v \) is identified by accessing the hash table of LIDBlocks on the SSD. If there are unused (NULL) slots in \( B \), one of the slots is overwritten with the LID of the new record. If there are no unused slots, then all LIDS in \( B \) and the LID of the new record are moved in bulk into the LID list in the index, amortizing the I/O cost.\(^3\)

In this model, index traversal is slightly more complex: all the non-NULL LIDs in the LIDBlock for a value also need to be treated as matches. Deletions and attribute value updates may also need to

\(^3\)One can shift the burden of LIDBlock flushing outside of running transactions by triggering an asynchronous flush once a LIDBlock becomes full (or nearly full), rather than waiting until it overflows.
traverse and modify a LIDBlock. There is some additional I/O, but only on solid state storage.

6.3.3 The Sparse Index Case

When there are few matches per index value, the organization above would need a very large number of LIDBlocks, most of which would be underutilized. Instead, for sparse indexes we maintain a single LIDBlock for an entire leaf node of the index. Rather than using a hash table hashed by attribute value, we use a hash table hashed by the address of the index leaf page. This address can be obtained using a partial traversal of the index, without accessing the leaf node itself. Since the internal nodes of a tree index occupy much less space than the leaf nodes, they are much more likely to be resident in the main memory bufferpool.

Searches have some additional overhead, because the shared LIDBlock would need to be consulted even for records that may not match the search condition. There would also be overhead during node splits and merges to maintain the LIDBlock structure.

The overhead of LIDBlocks on searches may be high for sparse indexes. For example, a unique index search would previously only have to look up one main file record. With a LIDBlock for a given key range, a search may need to read $b$ of them, where $b$ is the LIDBlock capacity. This example suggests an optimization: store both the LID and the key in the LIDBlock for sparse indexes. This optimization reduces the capacity of LIDBlocks, but significantly improves the magnetic disk I/O for narrow searches.

6.3.4 Revisiting Deletions and Updates

With LIDBlocks, it is now possible that an update or deletion occurs to a record whose LID is in a LIDBlock rather than the LID list. For deletions, one can simply set the LID to NULL in the LIDBlock. For updates that change the position of the record in the index, one needs to nullify the LID in the previous LIDBlock or LID-list, and insert it into the new LIDBlock.

In this way, LIDBlocks also improve the I/O behavior of updates to indexes on changed attributes. In the original scheme, HDD-resident index leaf pages appropriate to the new attribute value needed to be updated with the LID of the updated record. In the enhanced scheme, writes are needed only on SSD-resident LIDBlocks most of the time. Magnetic disk writes are amortized over the number of records per LIDBlock.

6.3.5 DeltaBlock Technique

To improve disk space utilization, we introduce DeltaBlock technique that exploits overlap among different versions of the same record. The key observation in managing and storing immutable versions of a record is that there tends to be a large overlap between consecutive versions of a record (typically only a small set of attributes is changed in transaction processing workload). The existing multi-version databases fail to exploit this overlap and redundantly store the unchanged portion of the record for every version. The result is growth of database size at much higher rate compared to the traditional single-version (using in-place update) data stores. In the worst case, the database size grows linearly with the average number of versions of each record.

To address this shortcoming, we develop a novel DeltaBlock technique to enable a natural and a fast data compression and decompression that exploits overlap among consecutive versions of a record and utilizes the SSD fast random access property. The key design feature of DeltaBlock is to avoid changing
the magnetic disk I/O pattern for neither query and update processing. We facilitate this I/O property by ensuring that the latest version of any record is retrievable with at most one disk and one SSD I/Os. In fact, we prove that one can do even better and query the latest $k$ versions of record with at most one disk I/O, a query that deemed up to $k$ I/O under traditional multiversion store when LIDBlock is not used.

Similar to our LIDBlock idea, and for each record represented by a LID, we batch in the SSD-resident DeltaBlock, the delta between consecutive versions of a record. And periodically, we flush a set of deltas for each record to disk upon overflowing of its corresponding DeltaBlock. But, more importantly, whenever deltas are flushed to disk, we also reconstruct the latest recent version of the records given a set of deltas, and we flush both the deltas and the latest record to disk. The flushing of periodically reconstructed version of the record fulfill our key design principle of retrieving any record with at most one HDD I/O. Also storing the deltas together with the latest reconstructed version of the record satisfies the second property of retrieving the latest $k$ versions of the tuple with at most one disk I/O.

One way to store these DeltaBlock is to extend the LtoR (LID,RID) to LtoR(LID, RID, DeltaBlock). Alternatively, DeltaBlock can be implemented as as a separate table or a hash table similar to LIDBlock data structure. The integration of DeltaBlock and Indirection techniques are pictorially presented in Figure 6.5, including the batching and flushing of deltas and flushing and reconstructing of latest version of a record.

---

4The cost of additional SSD I/Os is negligible w.r.t. the of HDD I/O.
6.4 Extended Example

In this section, we provide a detailed example to further illustrate the core of our Indirection proposal and motivate the analysis in Section 6.5.

Consider a table $R(A, B, C, D)$ with B-tree indexes on all four attributes. $A$ is the primary key, and is, therefore, sparse. $B$ is also sparse, while $C$ and $D$ are dense. $R$ is stored on disk in a versioned fashion. For each row we show the RID of the previous version (if any) of the row; the previous-RID may or may not be explicitly stored. Suppose that at a certain point in time, the extension of $R$ includes the rows given below. A flag indicating whether the row has been deleted is also included.

<table>
<thead>
<tr>
<th>RID</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Prev-RID</th>
<th>Deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>345</td>
<td>100</td>
<td>3732</td>
<td>3</td>
<td>5</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>367</td>
<td>120</td>
<td>4728</td>
<td>3</td>
<td>6</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>369</td>
<td>130</td>
<td>2351</td>
<td>2</td>
<td>5</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>501</td>
<td>100</td>
<td>3732</td>
<td>2</td>
<td>5</td>
<td>345</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose the $LtoR$ table for $R$ is given by

<table>
<thead>
<tr>
<th>LID</th>
<th>RID</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>367</td>
</tr>
<tr>
<td>11</td>
<td>369</td>
</tr>
<tr>
<td>13</td>
<td>501</td>
</tr>
</tbody>
</table>

Indexes use LIDs (e.g., 10, 11, 13) rather than RIDs to refer to rows, and only the most recent version is indexed. Given the above database, the immediate and deferred changes to the database in response to various update requests are described below. The I/O needed is summarized in square brackets (log I/O and I/O for index node splits/merges are ignored). Our I/O estimates assume that all internal index nodes are resident in the RAM bufferpool, and that all leaf nodes require I/O. These estimates also assume that all accesses to SSD-resident structures require I/O.\(^5\) The estimates also assume direct access to a page given the LID, as might be supported by a hash table. We assume that LiDBlocks contain $b$ LIDs, and that for sparse indexes we are storing both the key and the LID in the LiDBlock. We now describe precisely each key operation, breaking down its steps into immediate and deferred actions. Immediate actions must be completed within the operation itself. Deferred actions are those that can happen later, such as when an operation causes a page to become dirty in the bufferpool but the actual I/O write comes later.

**Update.** We update the row with key 100 such that attribute $D$ is changed from 5 to 6. The immediate actions are

- The LID of the row in question (i.e., 13) is identified using the index on $A$ [1 HDD read].
- The entry (13, 501) in the $LtoR$ mapping is read [1 SSD read].
- The row with RID 501 is read [1 HDD read].
- A new version of the record is created at the tail of $R$ with a new value for $D$ and a new RID (suppose 601).

\(^5\)This might be too pessimistic, particularly for LiDBlocks that could be small enough to be cached in RAM.
• An index traversal for key 5 is initiated on attribute \( D \). If LID 13 is present in the corresponding leaf, it is deleted; otherwise, LID 13 is located in the LIDBlock for key 5 and is removed from the LIDBlock. [1 HDD read, possibly 1 SSD read]

• A partial index traversal for key 6 is initiated on attribute \( D \), and LID 13 is inserted into the corresponding LIDBlock [1 SSD read].

The required deferred actions are summarized as follows.

• The data page containing the row with RID 601 is dirty and will need to be flushed to the HDD. [1 HDD write amortized over all modifications to the page\(^6\)].

• The entry (13, 501) in the LtoR mapping is changed to (13, 601) [1 SSD write].

• The index page containing the key 5 will be dirty if LID 13 was in the leaf. The dirty page will need to be flushed to the HDD [1 HDD write, amortized over all modifications to the page]. If LID 13 was in the LIDBlock for key 5 then the dirty LIDBlock will need to be flushed to the SSD [1 SSD write].

• The LIDBlock for key 6 is dirty and will need to be flushed to the SSD [1 SSD write].

**Insertion.** Consider the insertion of a new record (140, 9278, 2, 6). The resulting immediate actions are:

• The absence of a previous version of the row is verified using the index on \( A \), including the LIDBlock [1 HDD read, 1 SSD read].

• A new row is created at the tail of \( R \), with a new RID (suppose 654). The previous-RID field is NULL.

• A new LID (suppose 15) is allocated.

• For the two dense indexes on \( C \) and \( D \), the LIDBlocks for keys 2 and 6 (respectively) are identified, and LID 15 is inserted into each [2 SSD reads].

• For the two sparse indexes on \( A \) and \( B \), the LIDBlocks for keys 140 and 9278 (respectively) are identified using partial index traversals, and LID 15 is inserted (paired with the key) into each [2 SSD reads].

The deferred actions are as follows.

• The data page containing the row with RID 654 is dirty and will need to be flushed to the HDD [1 HDD write, amortized over all modifications to the page].

• The entry (15, 654) is inserted into the LtoR mapping [1 SSD read, 1 SSD write].

• The LIDBlocks for each of the four indexes are dirty, and need to be flushed [4 SSD writes].

• In the event that a LIDBlock fills (one time in \( b \) insertions), we need to convert all LIDs in the LIDBlock into regular index LIDs, and reset the LIDBlock to an empty state [4/b SSD writes, 3/b HDD reads (the leaf in the index on \( A \) has already been read), 4/b HDD writes].

---

\(^6\)Amortization is expected to be high on the tail of a table.
Deletion. Now suppose deleting the row with key 100, which results in the following immediate actions:

- The LID of the row in question (i.e., 13) is identified using the index on $A$ [1 HD read, possibly 1 SSD read].
- The pair $(13, 501)$ in the $LtoR$ table is located [1 SSD read].
- The row with RID 501 is read and the deleted flag is set to 1 [1 HDD read].
- LID 13 is removed from the leaf node of the index on $A$.

The deferred actions for deleting row with key 100 are

- The data page containing the row with RID 501 is dirty and will need to be flushed to the disk. [1 HDD write, amortized over all modifications to the page].
- The pair $(13, 501)$ in the $LtoR$ table is dropped [1 SSD write].
- The index leaf page for $A$ containing the key 100 is dirty and will need to be flushed to the HDD [1 HDD write amortized over all modifications to the page].
- Whenever one of the other indexes is traversed and LID 13 is reached, LID 13 will be removed from the corresponding LID list [1 extra HDD write to modify the leaf, amortized over all modifications to the page].

Search. Suppose that the search returns $m$ matches that all fit in one index leaf page of a sparse index. Since no write is involved, the search consists of only immediate actions:

- Traverse the index [1 HDD read].
- Read the $LIDBlock$ for the leaf node [1 SSD read].
- Map LIDs to RIDs [$m$ SSD reads].
- Read all matching records [$m$ HDD reads (assuming an unclustered table)].

<table>
<thead>
<tr>
<th>Technique</th>
<th>Type</th>
<th>Immediate SSD</th>
<th>Deferred SSD</th>
<th>Immediate HDD</th>
<th>Deferred HDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>Single-attr. update</td>
<td>0</td>
<td>0</td>
<td>$3 + k$</td>
<td>$\leq 2 + k$</td>
</tr>
<tr>
<td></td>
<td>Insertion</td>
<td>0</td>
<td>0</td>
<td>$1 + k$</td>
<td>$\leq 1 + k$</td>
</tr>
<tr>
<td></td>
<td>Deletion</td>
<td>0</td>
<td>0</td>
<td>$2 + k$</td>
<td>$\leq 1 + k$</td>
</tr>
<tr>
<td></td>
<td>Search Uniq.</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Search Mult.</td>
<td>0</td>
<td>0</td>
<td>$[m/I] + m$</td>
<td>0</td>
</tr>
<tr>
<td>Indirection</td>
<td>Single-attr. update</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>$\leq 2$</td>
</tr>
<tr>
<td></td>
<td>Insertion</td>
<td>$1 + k$</td>
<td>$2 + k + k/b$</td>
<td>1</td>
<td>$\leq 1 + (2k - 1)/b$</td>
</tr>
<tr>
<td></td>
<td>Deletion</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>$\leq 1 + k$</td>
</tr>
<tr>
<td></td>
<td>Search Uniq.</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Search Mult.</td>
<td>$[m/I] + m$</td>
<td>0</td>
<td>$[m/I] + m$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: Base vs. Indirection technique analysis. For single-attribute updates and deletions, we show the case where the LID was initially in the leaf rather than the $LIDBlock$. $n$ is the number of attributes, $k$ is the number of B-tree indexes, $m$ is the number of results returned by a search, and $I$ is the number of keys in an index leaf node.
6.5 Analysis

We analyze the performance of the Indirection method according to three criteria: (a) time for core operations; (b) SSD space requirements; (c) SSD wear-out thresholds.

6.5.1 Time Complexity

Table 6.1 summarizes the I/O complexity of the Base and Indirection methods. Most I/Os are random, meaning that SSD I/Os are much faster than HDD I/Os, by about two orders of magnitude on current devices. It is therefore worth investing a few extra SSD I/Os to save even one HDD I/O. Note that these estimates are pessimistic in that they assume that none of the SSD-resident data is cached in RAM. If commonly and/or recently accessed data was cached in RAM, many SSD read I/Os could be avoided, and many SSD writes could be combined into fewer I/Os.

The most striking observation is that with a small increase in SSD I/Os the costs of updates, insertions, and deletions are substantially reduced, while the cost of searches increases only slightly. With the Indirection technique, the immediate HDD cost is independent of the number of indexes.

6.5.2 SSD Space Complexity

We now estimate the space needed on the SSD for the LtoR table and the LIDBlocks. If \( N \) is the number of latest-version records in a table, then we need \( N \) (LID,RID) pairs in the LtoR table. In a typical scenario, table records may be 150 bytes wide, with 8 byte LIDs and 8 byte RIDs. Assuming a fudge-factor of 1.2 for representing a hash table, the LtoR table would constitute roughly 13% of the size of the current data in the main table.

We now consider the space consumption of LIDBlocks for sparse indexes; dense indexes would take less space. The number of LIDBlocks for a single sparse index is equal to the number of leaves in the index. With an 8 byte key, an 8 byte LID, an 8KB HDD page size and a 2/3 occupancy factor, a leaf can represent about 340 records. A LIDBlock contains \( b \) (LID,key) pairs, leading to a total space consumption of \( 16bN \times 1.2/340 \) bytes per index. Even a small value of \( b \), say 16, is sufficiently large to effectively amortize insertions. At this value of \( b \), the LIDBlocks for a single index constitute less than 1% of the size of the current data in the main table.

Thus, even with 7 indexes per table, the SSD space required in a typical scenario is only about 20% of the HDD space required for the current data. Taking the historical data into account, the relative usage of SSD space would be even lower. Given that SSDs are more expensive per byte than HDDs, it is reassuring that a well-provisioned system would need much less SSD capacity than HDD capacity.

6.5.3 SSD Life-Expectancy

SSDs based on enterprise-grade SLC flash memory are rated for about \( 10^5 \) erase cycles before a page wears out [129]. SSDs based on phase-change memory are rated for even higher rates. SSDs internally implement wear-leveling algorithms that spread the load among all physical pages, so that no individual page wears out early.

In flash-based SSDs there is a write-amplification phenomenon in which internal movement of data to generate new erase units adds to the application write workload. This amplification factor has been estimated at about 2 for common scenarios [71].
To estimate the wear-out threshold, suppose that the Indirection method uses an SSD page size of 512 bytes. Then a device with a capacity of $8 \times 10^7$KB can tolerate $8 \times 10^{12}$ writes before wear-out, assuming a write-amplification of 2 and $10^5$ erases per page.

Our most write-intensive operation is insertion, with about $1 + k$ SSD writes per insertion when there are $k$ indexes. Assuming nine indexes, and ten insertions per transaction, we would need 100 SSD writes per transaction. Even running continuously at the high rate of 800 transactions/second, the device would last more than three years.

### 6.6 Evaluations

We evaluate the performance of our proposed Indirection technique in three ways, to give complementary kinds of evidence in support of the technique. Since we are targeting operational data stores with both transactions and analytic queries, we base our evaluation on the TPC-H benchmark [203].

First, in Section 6.6.2 we try to answer the question “How would the performance of a state-of-the-art database change if it were to use our methods?” We employ the commercial database system DB2, but other database systems would have been equally good choices. The challenge is that we are not able to modify the DB2 source. Given a workload $W$, we construct a rewritten workload $W'$ for the DB2 engine that simulates the I/O behavior of our technique for $W$. While the workload $W'$ is not identical to $W$, we argue that the performance of DB2 on $W'$ provides a conservative estimate of the performance of $W$ on a (hypothetical) version of DB2 that implements the Indirection technique.

Second, in Section 6.6.3, we evaluate our Indirection technique by implementing it within the popular Generalized Search Tree (GiST) index package [104, 92]. The GiST package have successfully deployed into a number of well-known open-source projects including Postgres [4], PostGIS [3], OpenFTS [2], BioPostgres [1], and YAGO2 [5]. All aspects of the method (insertions, deletions, modifications) have been implemented; we refer to the resulting prototype as LibGiST$^{mv}$. We profile the I/O behavior of LibGiST$^{mv}$, and create a detailed I/O and execution cost model for the Indirection technique.

Finally, in Section 6.6.4, we shift our focus to TPC-H style analytical query processing that is geared towards an operational data store. We provide evidence for the key tenet of this chapter, namely, reducing the burden of index maintenance means that the system can afford more indexes, which in turn improves the performance of analytical query processing.

### 6.6.1 Platform

Experiments were run on two machines. The first machine, dedicated to our DB2 experiments, was running Fedora 8 and was equipped with a Quad-core Intel Xeon CPU E5345 running at 2.33 GHz, having 4GB of RAM, three magnetic disks (7200 RPM SATA), and one 80GB Single Level Cell (SLC) FusionIO solid state drive. The configuration of our second machine running Ubuntu 10.4, used exclusively for our LibGiST$^{mv}$ experiments, was a 6-core Intel Xeon X5650 CPU running at 2.67GHz, having 32GB of RAM, one magnetic disk (7200 RPM SATA), and one 80GB SLC FusionIO solid state drive.

7 Device characteristics are based on an 80GB FusionIO device.
8 Again, this estimate is pessimistic because it assumes no RAM caching of SSD pages.
9 Our SLC FusionIO can support up to 88,000 I/O operations per second at 50µs latency.
10 While the FusionIO devices are high-end devices that are relatively expensive, we remark that recent SSDs such as the Intel 520 series can store about 500GB, cost about $500, and can support 50,000 I/O operations per second at 85µs latency, more than sufficient for the workloads described here.
Table 6.2: Query re-writing to capture indirection mechanism

In our experiments, we used IBM DB2 version 9.7. We configured DB2 with adequate bufferpool size (warmed up prior to starting the experiments) to achieve an average 90% hit ratio on both data and index pages. For the DB2 experiments, we generate a TPC-H database [203] with scale factor 20. File system caching was disabled in all experiments.

For LibGiST™, we extended LibGiST v.1.0 to a multiversion generalized search tree C++ library that supports our Indirection techniques including LIDBlocks and DeltaBlocks.

6.6.2 DB2 Query Re-writing Experiments

The goal of the query re-writing experiment is to study the I/O pattern for both the unmodified DB2 system ("Base") and the Indirection approach. To evaluate Base for a query \( Q \) we simply run \( Q \) in the original schema \( S \). To evaluate Indirection we rewrite \( Q \) into another query \( Q' \). \( Q' \) is run in a schema \( S' \) containing an explicit LtoR table representing the LID-to-RID mapping. Ideally, the LtoR table is physically located on the SSD device; we empirically examine the impact of the location below. In \( S' \), base tables are augmented with an additional LID column, where the value of LID is generated randomly.

In \( S \) we build as many indexes as desired on the base tables. In \( S' \) we build a single index on the attribute selected in the query, typically the primary key of the base table.

Rewriting Queries

For queries, the rewriting simply adds the LtoR table to the FROM clause, with a LID equijoin condition in the WHERE clause. An example template of our query re-writing that simulates the indirection mechanism is shown in Table 6.2. The queries are written over TPC-H LINEITEM table and the indirection table, denoted by LtoR.

To see why the performance of \( Q' \) is a conservative estimate of the cost of the indirection technique, consider two cases for the query in Table 6.2. In the first case, some selection condition on an indexed attribute (or combination of conditions) is sufficiently selective that an access plan based on an index lookup is used. This case includes a point query specified as an equality condition on a key attribute. The Base plan for \( Q \) would include an index traversal and a RID-based lookup in the LINEITEM table. For \( Q' \), we will also have an index traversal and a RID-based lookup of LINEITEM, together with a LID-based lookup of the LtoR table. This is precisely the I/O pattern of the Indirection technique.

In the second case, the selection conditions are not very selective. In such a case, the system is likely
to scan the base table in answering $Q$.\footnote{We do not include old versions of records for these experiments; in a true multiversion database the base tables would contain some old record versions that would need to be filtered out during the scan, using the valid time attributes.} To answer $Q'$ in such a case requires a join of \texttt{LINEITEM} and \texttt{LtoR} in order to make sure that we only process the most recent versions of each record. This may actually an overestimate of the cost needed by the \texttt{Indirection} technique, because the \texttt{Indirection} technique can also employ a scan without consulting the \texttt{LtoR} table.

**Rewriting Updates**

For updates, an extra \texttt{UPDATE} statement is added to keep the \texttt{LtoR} table current, as illustrated in Table 6.2. Since we are simulating a multiversion database, the \texttt{Base} method inserts a new row rather than updating an existing row. While the \texttt{Base} method pays the cost of inserting the new row into each index, we are slightly favoring the \texttt{Base} method by ignoring the cost of deleting the old row from the indexes. Depending on the implementation technique used for temporal attributes (Section 6.1) there may be additional I/O required to update the temporal attributes of the old row, but we ignore such I/O here.

For updates in which a single attribute value is modified, the \texttt{Indirection} method incurs just one index update and one update to the \texttt{LtoR} table. At first it may seem like there is more work done by the \texttt{Indirection} method simulation for the updates of Table 6.2, since the \texttt{INSERT} statements are the same and the \texttt{Indirection} method has an extra \texttt{UPDATE} statement. This impression is true only for the case in which there is one base table index in the base schema $S$. As soon as there are multiple indexes on base tables in $S$, the cost of the \texttt{INSERT} statement for the \texttt{Base} method exceeds that of the corresponding \texttt{INSERT} in the \texttt{Indirection} method because more indexes need to be updated.

Our profiling of update statements can easily be extended to delete statements, but we omit such profiling because the performance profile would be similar to that for updates. On the other hand, our rewriting does not model the \texttt{LIDBlock} and \texttt{DeltaBlock} techniques, and thus cannot fully capture its performance advantages for insertions. The benefits of the \texttt{LIDBlock} and \texttt{DeltaBlock} techniques will be evaluated in Section 6.6.3.
Results

All DB2 measurements are averages over 5 million randomly chosen queries/updates with the exception of our selectivity experiments and analytical queries, in which fewer operations were performed to complete the experiments in a reasonable time.

Effect of Indexes on Execution Time. Our first result confirms that adding the extra indirection layer has negligible overhead for query processing as shown in Figure 6.6. For this experiment, queries are point queries that are accessed by specifying the value of the primary key attribute. The query overhead is negligible because the indirection mapping from LID-to-RID requires only a single additional random SSD I/O, a delay that is orders of magnitude faster than the necessary random magnetic disk I/O for retrieving data pages holding entire records. Figure 6.6 also shows that the update execution time for the Base technique increases dramatically as the number of indexes is increased. With 16 indexes, Indirection outperforms Base by a factor of 3.6.

Digging deeper into the bufferpool behavior of DB2 also reveals that with a modest number of SSD page writes, the number of magnetic disk index writes are substantially reduced, as shown in Figure 6.7. This result demonstrates the effectiveness of Indirection in reducing the index maintenance cost for multiversion databases.

Effect of Query Selectivity and Index Clustering. Consider a range query over a single indexed attribute in the LINEITEM table. By varying the width of the range, we can vary the query selectivity. Figure 6.8 shows the results for various selectivities, where the indexed attribute is the key of the LINEITEM table by which the table is clustered. On average the query overhead (for consulting the LtoR table) remains under 20% as query selectivity varied. There is a sudden jump at around 0.22% selectivity, but a closer examination of the DB2 query plans reveals that the sudden increase in execution time is attributable to the optimizer incorrectly switching from a nested-loops to a merge-sort join plan.

The 20% overhead, while still small, is higher than the negligible overhead seen for point queries.

---

12 The starting value of the range in our range queries are chosen randomly.
13 This behavior can be avoided if optimizer hints are provided.
in Figure 6.6. We also observed that as the index clustering ratio decreases, the query processing gap between Indirection and Base decreases. For example, for the lowest clustering ratio index of the LINEITEM table (on the SUPPKEY attribute), the overhead drops to only 4% on average. These differences can be understood as a consequence of caching. With a high clustering ratio, one base table disk page I/O will be able to satisfy many access requests, meaning that each magnetic disk I/O is amortized over many records. On the other hand, every record will contribute an SSD I/O for the LtoR table since that table is not suitably clustered. For point queries and queries over an unclustered index, the ratio of SSD I/O operations to magnetic disk I/Os will be close to 1.

Indirection Mapping Implementation. Finally, we illustrate that the indirection random read/write access pattern is ideal for SSDs and not for magnetic disks. We tried two different implementations of the LID-to-RID mapping using either a DB2 range-clustered table (RCT) or a traditional B-Tree index hosted on either SSDs or HDDs. As shown in Figure 6.9, when an SSD is used, the overall query and
update cost is 1.9X and 2.7X lower, respectively, than on an HDD.

6.6.3 GiST Implementation

Our LibGiST\textsuperscript{mv} codebase directly implements all aspects of our proposed approach, including the \textit{LID-Block} and \textit{DeltaBlock} techniques. We employ LibGiST\textsuperscript{mv} as the basis for a systematic performance study in a controlled environment. All HDD and SSD reads and writes used \textit{Direct I/O} to avoid extra copying of operating-system buffers. No bufferpool space was allocated for the indirection table, so requests to the indirection table always incur SSD I/Os.

In our prototype, we also extended LibGiST bufferpool in order to test a variety of memory configurations. We further enhanced the LibGiST library to collect statistics on index operations, file operations, and a detailed bufferpool snapshots. All prototype experiments used the TPC-H schema with scale factor 1, and the workload consisted of random point queries and random insert and update queries (conceptually similar to the workload presented in Section 6.6.2). We focus on the I/O profile of index traversals and updates as encountered by queries, updates, and insertions. All results are averaged over $10^5$ random queries/updates/insertions.

**Comparison of Average Execution Time.** We first isolate the query, update, and insert execution times for a single index with a small bufferpool size, enough pages to pin the root of the index and all pages in one path from the root to a leaf. Figure 6.10 shows that the insert and query times are virtually the same due to the negligible overhead introduced by the additional SSD read and write I/O operations. The “Update” column for the \textit{Indirection} method in Figure 6.10 reflects a traversal of the index to locate the prior version of the record, plus an update of the SSD-resident LID-to-RID mapping. The base method is more expensive because it has to perform additional writes to the magnetic disk.

**Multiple Indexes.** The update time shown in Figure 6.10 does not capture the true benefit of the \textit{Indirection} method when there are multiple indexes. In such a case, the \textit{Indirection} method needs to traverse only one index, and update one LID-to-RID mapping.\textsuperscript{14} In contrast, the base method needs

\textsuperscript{14}If an indexed attribute is updated, then extra I/O is needed for that index to relocate the index entry. Indexes that are neither traversed nor updated in the \textit{Indirection} method are said to be “unaffected.”
to traverse and update HDD-resident pages for every index. Figure 6.11 shows that the performance improvement approaches 20X as the number of indexes increases from 1 to 16.

**Varying the Bufferpool Size.** We consider five categories of bufferpool sizes large enough to hold: (1) only few pages (small), (2) all index non-leaf pages, (3) the entire index, (4) the entire index and all data pages, or (5) everything including the index, data, and LtoR table. These sizes reflect possible use cases where the system has memory ranging from very small to very large. We explored an update workload under these five settings when having one (Figure 6.12(a)) or sixteen (Figure 6.12(b)) indexes on unaffected attributes. Note the logarithmic vertical scale.

In Figure 6.12(a), only when both the index and data pages are memory-resident, but the LtoR table is not, does the Indirection method perform poorly compared to the Base approach. This is not surprising, because even a fast SSD I/O is slower than access to RAM. When an update operation results in some HDD I/O (either due to updating leaf pages or data pages), then Indirection is superior by up to 2X. When everything including the LID-to-RID mapping table is memory-resident, then Indirection continues to be superior because it does not touch most of the indexes.

Again, the true benefit of the Indirection technique surfaces as we increase the number of indexes. For example, when scaling the number of indexes to sixteen in Figure 6.12(b), Indirection typically wins by an order of magnitude. These experiments demonstrate that the Indirection technique provides significant benefits even if large portions of the database are memory-resident.

**Varying the LIDBlock Size.** So far our focus has been on improving index maintenance involving updates. We now demonstrate the power of our LIDBlock approach for insertions. In Figure 6.13, we vary the capacity of LIDBlock from no LIDs to 256 LIDs for the non-unique index defined on the suppkey attribute of the lineitem table. When increasing the LIDBlock size to around 32 LIDs, we observed that the insertion cost is significantly reduced by up to 15.2X. This improvement is due to the amortization of a random leaf page update over many insertions, e.g., a LIDBlock size of 32 results in batching and flushing to the disk once every 32 insertions on average.

We can demonstrate a similar benefit with a non-unique index having many more duplicate entries, such as an index on the quantity attribute of the lineitem table, having 50 distinct values and 0.2M records per value. It is beneficial to allow larger LIDBlock sizes as shown in Figure 6.14, in which the
insertion execution time is reduced by up to 4.9X. Unlike the previous case, the main reason for the speedup is not simple amortization of insertions; since there are so few key values the tree structure of the index is shallow and its traversal is already cheap due to caching. Instead, the speedup is due to having batches large enough to be resolved with one or two I/Os to update the LID lists for a key.

**Effect of Varying DeltaBlock Size.** Our final prototype experiment demonstrate the role of our DeltaBlock technique. In this experiments, we begin an index defined over `lineitem` table with also scaling factor 1 followed by randomly updating 100k randomly selected records 12 times. As we scale the size of LIDBlock from 0 to 9, we compute the HDD usage (raw data size), SSD usage (LIDBlock size), and relative query and update execution time.

As demonstrated in Figure 6.15, when increasing the LIDBlock size, we obtained a compression reduction ratio of 80% with as few as 9 lids per DeltaBlock. This reduction is made possible by allocating to LIDBlock only 6% of the size of uncompressed raw data; hence, exhibiting an SSD usage that is only 6% of HDD usage. In addition, we also illustrated that query processing (on the latest record version) cost is nearly unaffected by introducing DeltaBlock in fact, we also observed an improved update cost due to
compressed table. But, more importantly, using the \texttt{DeltaBlock}, one can retrieve the latest \( k \) versions of record with a single HDD I/O as oppose to the \( k \) required random I/Os by \texttt{Base} technique.

### 6.6.4 DB2 Operational Data Store Experiments

In this section, we study the effects of adding indexes in the context of an operational data store, in which small-scale analytical query processing is competing with transactional throughput. Our query workload is based on prior work \cite{40} that modifies TPC-H queries so that they each touch less data. For our index workload, we rely on the DB2 Index Advisor recommendation given our query workloads.

We first consider only the primary key indexes of the TPC-H relations. Subsequently, we add the remaining indexes recommended by DB2 Advisor one at a time, starting from the most to least beneficial index. After each round of index addition, we re-run our query workload. Likewise, after adding each index, we compute the update cost for a uniformly generated update workload, in which each non-
primary-key attribute has an equal chance of being updated. The update cost is a normalized average execution time of updating indexes on DB2 and our on LibGiST\textsuperscript{mv} with bufferpool size set to either small and non-leaf pages. The results are summarized in Figures 6.16 and 6.17.

Our first observation is that analytical query time is substantially reduced (by a factor of 3) as we add more indexes recommended by DB2 Advisor. More importantly, we observe that the additional indexes are more “affordable” for updates because our Indirection technique reduces the index maintenance overhead. In the base configuration, the index maintenance overhead increases linearly as more indexes are added, reducing transaction throughput. Our Indirection technique reduces the update cost by more than 40%.

Figure 6.17 shows a two dimensional plot of relative query time $q$ versus relative update time $u$. On both axes, smaller is better. Each index configuration determines a point $(u, q)$ on this graph, and one can choose a suitable configuration to achieve a desired query/update trade-off. In Figure 6.17, the
Indirection technique dominates the Base method. This is a key result: The Indirection technique makes indexes more affordable, leading to lower query and update times.

To understand the importance of Figure 6.17 consider the following scenarios.

1. A DBA has an update-time budget of 0.6 units, and within that budget wants to optimize query processing time. According to Figure 6.17, the Indirection technique can achieve query performance of about 0.32 units under such conditions, while the Base method can achieve only 1.0 units, three times worse.

2. A DBA has a query-time budget of 0.5 units, and within that budget wants to optimize update-time. The Indirection technique can achieve update performance of about 0.5 units under such conditions, while the Base method can achieve only 0.7 units, 40% worse.

3. A DBA wants to minimize the sum of update-time and query-time. The Indirection method can achieve a sum of about 0.87 at (0.55,0.32), whereas the best configuration in the Base method is 1.15 at (0.75,0.4), 32% worse.

6.7 Summary

The multi-version temporal database market is growing [53]. A temporal database simplifies application development and deployment by pushing the management of temporal logic into database engines. By adopting temporal technologies, the development cost can be reduced by a factor of 10 [53]. This success has led major database vendors (including Oracle [162], IBM [110], and TeraData [193]) to provide support for multiversion temporal data.

We tackle a key challenge of multiversion databases: providing good update performance and good query performance in a single system. Transaction throughput and analytical query processing often have conflicting requirements due to the high index maintenance cost for transactions. Our efficient index maintenance using Indirection makes indexes more “affordable,” substantially improving the available

---

15Except when a very small weight is assigned to update time.
configuration choices. Our evaluation demonstrates a query cost reduction by a factor of 3 without an increase in update cost. The batching of insertions using our LIDBlock technique can save up to 90% of the insertion time.
Part III

Kernel Extensions
Chapter 7

XML/XPath-to-Kernel Mapping

7.1 Introduction

In XML filtering, namely, selective information dissemination (SID), the information content is represented as an XML document (an event) while the filters are expressed as XPath expressions/queries (subscriptions) [167, 95, 67, 45, 98, 37, 106]. The existing techniques in XML filtering are designed to detect overlap among a set of XPath expressions in order speed up the filtering of incoming XML documents. But the overlap detection algorithms are restricted to finding common prefixes and other possible forms of overlap such as suffix and infix are largely unexplored [175]. Furthermore, existing filtering algorithms are mostly limited to NFA- or DFA-based processing models, although expressive, this choice substantially hinders the overall performance, as shown in [175].

The XML filtering problem is conceivably identical to the publish/subscribe (stateless) matching problem discussed in Section 3.2. This reformulation is possible if XPath expressions and XML documents are to be mapped into Boolean expressions and attribute-value pairs, respectively. In this chapter, we present a mapping of XML filtering to the pub/sub matching problem in order to leverage the abundance of existing pub/sub matching algorithms [220, 12, 74, 94, 18, 211, 176, 82, 178]. The main distinguishing strength of existing pub/sub matching algorithms (i.e., our query indexing kernel) are detecting arbitrary overlaps among Boolean expressions, namely, the common predicates that are not limited to only prefixes, and storing and matching the common predicates exactly once [74, 211, 176, 175].

This chapter focuses on extending and formalizing the XML/XPath encoding for achieving high-throughput XML filtering engines; the effectiveness of our proposal was evaluated in [175]. The key contributions of this chapter are (1) to propose an XML/XPath encoding that enables vertical and horizontal navigation and structural and value-based filtering within an XML document (cf. Sec. 7.2); (2) to formalize an XPath language encoding and to develop a dichotomy of the supported class of XPath expressions vs. the unsupported class of XPath expressions (cf. Sec. 7.3-7.4).

7.2 XPath Expression Encoding

In this section, we present our predicate calculus for mapping a subset of XPath expressions into Boolean expressions (i.e., subscriptions), namely, a conjunction of Boolean path-consistent predicates. Furthermore, we present our XML document encoding into a set of attribute-value pairs (i.e., events). The
encoding of XPath expressions and XML documents enables reformulating the XML filtering problem as stateless matching problem outlined in Section 3.2.

In general, an XPath expression enables navigation through an XML document (having a tree structure) either by moving from elements of a document to their subelements (XPath vertical axis) or moving from elements of a document to their sibling elements (XPath horizontal axis). Through this navigational power, the XPath expression defines a pattern for selecting relevant parts of XML documents, and if an XPath expression selects a non-empty part of an XML document, then the XML document is considered matched. The XPath expression is defined as a sequence of location steps over nodes \((\cdots/n_{i-1}/n_i/\cdots/n_l)\). For example, an absolute XPath expression is evaluated starting from the root of the document and each node \(n_i\) is applied on the result of its previous node \(n_{i-1}\). Alternatively, each XPath expression can be viewed as a conjunction of Boolean predicates, which is evaluated over document paths, where the \(i^{th}\) predicate is derived directly from the node of \(i^{th}\) location step in the XPath expression.

We consider XPath expressions that impose constraints on both absolute and relative paths within an XML document and support constraints on both horizontal and vertical positioning and relative structural positing of each node within the XML document (i.e., XML tree). Each of these constraints are expressed through our proposed predicate calculus (extending the formalism in [175]).

XPath expressions \((n_1/n_2/\cdots/n_l)\) and \((/n_1/n_2/\cdots/n_l)\) are defined as relative and absolute location paths, respectively, which is a sequence of location steps separated by / or //. Each location step may consists of an axis, a node test, and zero or more attributed-based filters.

We define an XML document \(D\) as directed tree which consists of a set of paths, \(D = (p_1,\ldots,p_l)\), stretching from the root of the XML tree to each leaf. Each path \(p_i\) consists of a set of node elements, and each may contain attributes and values and content elements.

**Vertical Constraints**

To express the downward vertical constraint, we define the Boolean predicate

\[
\downarrow P_{n,\text{opt},\text{val}},
\]

(7.1)

where \(\downarrow P\) denotes the predicate for downward vertical constraints, \(n\) is the node on which the vertical constraint is imposed (i.e., \(n\) is equivalent to attribute/dimension in our formalism in Section 3.2), \(\text{opt} \in \{=,\geq\}\), and \(\text{val} \in \mathbb{N}\). The Boolean predicate \(\downarrow P\) represents constraint on downward vertical position of the node \(n\). The predicate \(\downarrow P\) is satisfied for a given XML document if and only if there exists a path \(p_i\) in the XML document, where the path has the node \(n\) at depth \(\text{val}'\) such that \((\text{val}' \text{opt} \text{val})\). The predicate \(\downarrow P\) is primarily used to encode the vertical positioning of the first location step in an XPath expression not starting with a wildcard.

Similarly to express the upward vertical constraint, we define the Boolean predicate

\[
\uparrow P_{n,\text{opt},\text{val}},
\]

(7.2)

where \(\uparrow P\) denotes the predicate for upward vertical constraints, \(n\) is the node on which the vertical constraint is imposed, \(\text{opt} \in \{\geq\}\), and \(\text{val} \in \mathbb{N}\). The Boolean predicate \(\uparrow P\) represents constraint on upward vertical position of the node \(n\). The predicate \(\uparrow P\) is satisfied for a given XML document if and

\footnote{We suppose that root of XML tree has a depth of 1.}
only if there exists a path $p_i$ with length $l$ in the XML document, where the path has the node $n$ at depth $\text{val}'$ such that $((l - \text{val}') \text{ opt val})$. The predicate $\downarrow P$ is primarily used to encode the vertical positioning of an XPath expression ending in a wildcard.

**Horizontal Constraints**

We define two Boolean predicates to impose constraints on the horizontal positioning for each node relative to their immediate siblings (those sibling with whom they have a common immediate parent). These predicates enable specifying forward and backward positioning. To express forward horizontal constraint, we define

$$\Rightarrow P_n, \text{opt}, \text{val},$$

while for backward horizontal constraint, we define

$$\Leftarrow P_n, \text{opt}, \text{val},$$

where $\Rightarrow P$ denotes the predicate for forward horizontal constraints, $\Leftarrow P$ denotes the predicate for backward horizontal constraints, $n$ is the node on which the horizontal constraint is imposed, $\text{opt} \in \{<, \leq, =, \geq, >\}$, and $\text{val} \in \mathbb{N}$. For example, $\Rightarrow P_{n,=,2}$, implies that the node $n$ has one immediate sibling to its left, while $\Leftarrow P_{n,=,2}$, implies that the node $n$ has one immediate sibling to its right.

**Relative Vertical Constraints**

To express the relative vertical constraint, we define the Boolean predicate

$$\uparrow P(n_1, n_2), \text{opt}, \text{val},$$

where $\uparrow P$ denotes the predicate for relative downward vertical constraints, $n_1$ and $n_2$ are the two nodes on which the relative vertical constraint is imposed, $\text{opt} \in \{=, \geq\}$, and $\text{val} \in \mathbb{N}$. The Boolean predicate $\uparrow P$ represents constraint on relative downward vertical position of the nodes $n_1$ and $n_2$. The predicate $\uparrow P$ is satisfied for a given XML document if and only if there exists a path $p_i$ in the XML document, where the path contains both nodes $n_1$ and $n_2$, at depth $\text{val}_1$ and $\text{val}_2$, respectively, such that $((\text{val}_2 - \text{val}_1) \text{ opt val})$. Essentially, imposing a constraint on a vertical distance of any two nodes. The predicate $\uparrow P$ is primarily used to encode the relative vertical position constraint on consecutive locations steps in an XPath expression.

**Relative Structural Constraints**

To express the relative structural property constraint, we define the Boolean predicate

$$\text{axis} P_{n, \text{opt}, \text{val}},$$

where $\text{axis} P$ denotes the predicate for relative structural property constraints, $n$ is the node on which the constraint is imposed, $\text{opt} \in \{<, \leq, =, \geq, >\}$, and $\text{val} \in \mathbb{N}$. We support any $\text{axis}$ chosen from the set $\text{child}, \text{parent}, \text{descendant}, \text{ancestor}, \text{following}, \text{following-sibling}, \text{proceeding}, \text{and proceeding-sibling}$. 
For instance, the predicate \( \text{ancestor}^{m \leq 3} \), is satisfied if and only if the node \( n \) has at least 3 ancestors along its path from the root.

**Attribute-based Constraints**

To express the attribute-based filters, we define the Boolean predicate

\[
\mathp@{P}(n, \text{attr}, \text{opt}, \text{val}), \tag{7.7}
\]

where \( \mathp@{P} \) denotes the predicate for attribute-based constraints, \( \text{attr} \) is the attribute of the node \( n \) on which the constraint is imposed, \( \text{opt} \in \{<, \leq, =, \geq, >\} \), and \( \text{val} \in \mathbb{N} \). In short the predicate \( \mathp@{P} \) enables imposing a constraint on any node attributes in a given XML document.

**Path-length Constraints**

To express the path-length constraint, we define the Boolean predicate

\[
\mathp{i}{P}(\text{opt}, \text{val}), \tag{7.8}
\]

where \( \mathp{i}{P} \) denotes the predicate for path-length constraints, \( \text{opt} \in \{\geq\} \), and \( \text{val} \in \mathbb{N} \). The Boolean predicate \( \mathp{i}{P} \) imposes a constraint on the minimum length of a relevant path in an XML document; thereby, this predicate operates on a path as a whole and does not require a node name. The predicate \( \mathp{i}{P} \) is satisfied for a given XML document if and only if there exists a path \( p_i \) with length \( l \) in the XML document such that \( (l \text{opt} \text{val}) \). The predicate \( \mathp{i}{P} \) is necessary to encode XPath expressions consisting of only wildcards.

### 7.3 XPath Encodings

In what follows, we present how various XPath language features are expressed in our predicate calculus by extending the formalism in [175]. We conclude by formalizing the subset of the XPath language that is supported by our encoding.

#### 7.3.1 XPath Expressions

First, in order to identify repeating nodes (having the same name) in an XPath expression, each node name is associated with a unique occurrence number (written as superscript). For example, the XPath expression \((n_1/n_2/n_3/n_1)\) is represented as \((n_1^1/n_2^1/n_3^1/n_1^2)\).

**Encoding of Simple XPath Expressions**

In our formalism, we define a *simple* XPath expression as an expression that consists of only child axis (/) and does not contain any other axiom such as descendant (/\/) or other operators such as wildcards (*). Thus, a *simple* absolute XPath expression is given by

\[
/n_1/.../n_1. \tag{7.9}
\]
Under our predicate calculus, it is mapped into a conjunction of vertical constraints (i.e., Boolean predicates) as follow:

\[
(\downarrow P_{n_1}^{=1}) \land \left( \bigwedge_{i=2}^{l} (\downarrow P_{n_{i-1},n_i}^{=1}) \right)
\]  
(7.10)

The first Boolean predicate, \( (\downarrow P_{n_1}^{=1}) \), ensures that the XPath expression matches the root of XML tree, namely, the root of XML document must be the node \( n_1 \), while the subsequent Boolean predicates dictate the relative distances between pairs of consecutive nodes of the matching path in the XML tree.

Similarly, a simple relative XPath expression is defined as

\[
\text{node}^{1}/\cdots/\text{node}^{m+1}
\]

and based on our predicate calculus, this simple expression is also mapped into a conjunction of vertical constraints (i.e., Boolean predicates) defined as

\[
(\downarrow P_{n_1}^{\geq1}) \land \left( \bigwedge_{i=2}^{l} (\downarrow P_{n_{i-1},n_i}^{=1}) \right)
\]  
(7.12)

Unlike the absolute XPath expression encoding, the first Boolean predicate in the relative path, \( (\downarrow P_{n_1}^{\geq1}) \), only ensures XML tree contains the node \( n_1 \) and that the depth of the node \( n_1 \) is greater or equal to 1. In another words, the node \( n_1 \) must exist in any matching path of the XML document. Interestingly, one can observe that the second Boolean in the relative expression encoding also requires that node \( n_1 \) exist and precedes \( n_2 \); thus, as long as relative path consists of two location steps, the Boolean predicate that ensures the existence of node \( n_1 \) can be dropped. We can conclude that the only difference between a relative and an absolute XPath encoding is the extra Boolean predicate required in absolute XPath for matching the root node. Without loss of generality, in the reminder of this section, we focus only on the relative path for the ease of presentation unless stated otherwise.

**Encoding of XPath Expressions with Wildcards**

A relative (or an absolute) XPath expression may start with \( k \) wildcards, end in \( k \) wildcards, and/or contain \( k \) wildcards anywhere in the middle.\(^2\) This general form is given by

\[
* / \cdots / * / \text{node}^{1} / \cdots / \text{node}^{m+1} / * / \cdots / * / \text{node}^{l-m} / * / \cdots / *
\]

In order to encode this XPath expression in our predicate calculus, we must account for these three general cases. The sequence of nodes which does not contain any wildcards are essentially simple XPath expressions (either relative or absolute), which we can encoded using Equations 7.10 and 7.12.

To encode expressions starting with \( k \) wildcards, we can utilize the downward vertical constraint, given by

\[
(\downarrow P_{n_1}^{\text{opt,m+1}})
\]

where \( \text{opt} \) is \( \geq \) and \( = \) for relative and absolute path, respectively. Similarly, for encoding the expression ending with \( k \) wildcards, we rely on the upward vertical constraint as follows:

\(^2\)Any number of \( k \) consecutive wildcards sequence can appear within an XPath expression.
Lastly, to express $k$ consecutive wildcards in the middle of expressions, we use the relative vertical positioning as follows:

\[ \uparrow P(n_i, n_{i+m+1}), \text{opt}, m, \geq 1, \] 

(7.16)

where the nodes $n_i$ and $n_{i+m+1}$ are any pair of nodes that are separated by $k$ wildcards.

In addition, our predicate calculus seamlessly extends to other interpretation of wildcards, including $?$ (matches 0 or 1 nodes), $\ast$ (matches 0 or more nodes), and $+$ (matches 1 or more nodes). For example, the relative vertical constraint can easily be adjusted to account for different semantics: $\uparrow P(n_i, n_{i+2}), \leq 1$, $\uparrow P(n_i, n_{i+2}), \geq 0$, and $\uparrow P(n_i, n_{i+2}), \geq 1$, for $?$, $\ast$, and $+$, respectively.

Encoding of XPath Expressions with Descendant

XPath expressions containing descendant operators (/\/) are easily supported in our formalism. Basically the only difference between / and // is simply the operator in the vertical constraint. Any pair of consecutive nodes $n_i/n_{i+1}$ is encoded as

\[ \uparrow P(n_i, n_{i+1}), \geq 1, \] 

(7.17)

In contrast, as shown before, $n_i/n_{i+1}$ is encoded as

\[ \uparrow P(n_i, n_{i+1}), = 1, \] 

(7.18)

Encoding of XPath Expressions with Positional Predicates

Positional predicates enable navigation to particular elements within an XML document or test a condition with respect to the current horizontal position of the node. Consider the following relative XPath expression:

\[ n_1/\ldots/n_{i-1}/n_i[position()] = \text{val}]/\ldots/n_i, \] 

(7.19)

where $\text{val} \geq 1$. The $position()$ enables navigating to and selecting the $\text{val}^{th}$ child of the node $n_{i-1}$. Again, such constraints can easily be expressed in our predicate calculus by leveraging the following predicate.

\[ \Rightarrow P n_i = \text{val} \] 

(7.20)

Alternatively, the position predicate can operate in a reverse horizontal direction from right-to-left as opposed to left-to-right direction. The reverse positioning can be achieved by referencing the $last()$ function as follows:

\[ n_1/\ldots/n_{i-1}/n_i[position()] \text{ opt (last() - val)}/\ldots/n_i, \] 

(7.21)

where $\text{opt} \in [\lt, \leq, =, \geq, \gt]$, which imposes a condition on the horizontal position of the $n_{i-1}$ child selected by the XPath expression, namely, selecting those child nodes whose positions satisfy the condition
[position() opt (last() − val)]. This condition is expressed in our predicate calculus as

\[ \iff p_{n_i, \text{opt}(1+\text{val})}. \] (7.22)

**Encoding of XPath Expressions with Axiom Predicates**

The axiom predicate is required to capture the horizontal and vertical structural conditions in XPath expressions. Consider the following XPath expression that imposes a structural condition:

\[ n_1/.../n_{i-1}/n_i[@(\text{attr}) \text{opt val}]/.../n_l. \] (7.25)

where \text{axis} is chosen from the set \text{child}, \text{parent}, \text{descendant}, \text{ancestor}, \text{following}, \text{following-sibling}, \text{proceeding}, and \text{proceeding-sibling} (cf. Figure 7.1). Therefore, the predicate \( \text{count}(\text{axis}) \) imposes certain structural conditions with respect to the current node \( n_i \), e.g., the condition \( n_i[\text{count(descendant)} = 4] \) is satisfied if and only if the node \( n_i \) has exactly four descendants. The axiom predicates are expressed in our predicate language as

\[ \text{axis} p_{n_i, =.\text{val}}. \] (7.24)

**Encoding of XPath Expressions with Attributed-base Filters**

Consider the following XPath expression with an attribute-based filter

\[ n_1/.../n_{i-1}/n_i[@(\text{attr}) \text{opt val}]/.../n_l. \] (7.25)

Similar to our axiom and positional predicates, the attribute-based filter for node \( n_i \) is simply encoded as

\[ @ p(n.\text{attr}).\text{opt}.\text{val}. \] (7.26)
Chapter 7. XML/XPath-to-Kernel Mapping

7.3.2 XPath encoding illustration

The following exemplifies our XPath expression encoding that capture the key aspects of our predicate calculus.

\[
X_1: */b/c \\
S_1: p^b_1,=2 \land \#^p(b_1,c_1),=1 \\
X_2: */a/*/b/c \\
S_2: p^a,\geq,2 \land \#^p(a_1,b_1),\geq,2 \land \#^p(b_1,c_1),=,1 \\
X_3: /a/a/*/a \\
S_3: p^a,-,1 \land \#^p(a_1,a_2),=,1 \land \#^p(a_2),\geq,5 \\
X_4: c/a/b/*/a \\
S_4: \#^p(c_1,a_1),=,1 \land \#^p(a_1,b_1),=,1 \land \#^p(b_1,a_2),\geq,4 \\
X_5: a/b[count(child) \geq 3]/a \\
S_5: \#^p(a_1,b_1),=,1 \land \#^p(a_1,b_1),\geq,3 \land \#^p(b_1,a_2),\geq,4 \\
X_6:/*//*//*//*/* \\
S_6: p,\geq,3
\]

Notably, there are common predicates among the above XPath expressions, which are fully exploited in pub/sub matching algorithms [74, 175]. For instance, there exist an overlap between XPath expressions \(S_4\) and \(S_5\) because \(S_4\) predicates are a subset of \(S_5\), whereby resulting in storing only unique predicates and avoid re-evaluation of duplicate predicates.

7.3.3 XPath encoding dichotomy

In this section, we formalize the subset of XPath language features supported by our encoding and other important properties of our encoding. Before, proceeding, we need to define a few necessary terms in order to precisely describe an XML document and an XPath expression.
Definition 9. An XML document is represented as (directed) tree structure.

Definition 10. A complete path in an XML document (tree) is a directed path from the root to a leaf.

Based on the definition of complete path, we can define paths that are crucial to our encoding, namely, valid paths.

Definition 11. A valid path in an XML document (tree) is a subset of elements in a single complete path.

To further clarify, we have provide an example of an invalid paths (paths that are not supported by our encoding) in Figure 7.2.

Definition 12. Lastly, an XPath expression is path-consistent, if it navigates and selects elements along a single path, i.e., selects subsets of elements from a valid path.

Now, we are in a position to present the dichotomy of supported and unsupported classes of XPath expressions.

Theorem 12. Any path-consistent XPath expression can be encoded as a conjunction of path-consistent predicates.

Proof. The path-consistency proof follows our matching operational semantics, in which an XML document is, first, transformed into a set of complete paths and each complete path is individually matched against all XPath expressions. Therefore, if an XPath expression is path-consistent, then it is sufficient to examine every complete path in order to determine whether an XPath expression matches the XML document. Examining each path individually is possible because a path-consistent XPath expression always navigates and operates on a single complete path.

Most importantly, the size complexity of our XPath encoding further justifies the effectiveness of our approach.

Theorem 13. The XPath encoding grows linearly with respect to the size of an XPath expression.

Proof. Each location step (or its associated predicate) in the XPath expression is encoded as a single predicate to incorporate the structural and filtering conditions associated with each node such as the vertical and horizontal positions of each node in XPath expressions, wildcards and descendant operators, and the axes predicates. Hence, the encoding size is linear with respect to the size of the XPath expression.

7.4 XML Document Encoding

The two pillars underlying our reduction of XML/XPath matching into attribute-value pair matching are the encoding of XML documents (into events) and XPath expressions (into subscriptions). In the previous section, we dealt with the latter problem. We now shift our focus to the encoding of an XML document into a set of attribute-value pairs, i.e., events – extending the XML encoding in [175]. Alternatively, one could use the formalism in Section 3.2, which also formulates events as Boolean expressions, which is provably more expressive. Therefore, using the latter formalism, the events will be
mapped based on our predicate calculus as opposed to sets of attribute-value pairs, a direction, which we will not pursue for the ease of presentation.

As described previously, each XML document \( \mathcal{D} \) is viewed as a rooted directed tree that can be represented as a set of paths, \( \mathcal{D} = \{ p_1, ..., p_l \} \), starting from the root node to each leaf node. Each path \( p_i \in \mathcal{D} \) consists of set of ordered nodes

\[
p_i = \{ n_1, \cdots, n_l \}.
\]  

(7.27)

Additionally, each node may be associated with a set of attributes or content elements. Furthermore, the node \( n_j \) is at depth \( j \) in the XML tree. Without loss of generality, we focus on mapping paths (set of nodes and not attribute elements) into a set of attribute-value pairs. In what follows, we first assume only distinct node names in each path, if nodes are not distinct (as shown later), then they can be made unique by adding occurrence numbers, which was introduced in Section 7.3. Therefore, the path \( p_i \) is mapped as follows:

\[
\bigcup_{i=1}^{l} (n_i, i)_{\downarrow} \cup (n_i, \text{for.pos})_{\Rightarrow} \cup (n_i, \text{back.pos})_{\Leftarrow} \cup \bigcup_{i=1}^{l} \bigcup_{j=1}^{|\text{axis}|} (n_i, \text{count}(\text{axis}_j))_{\text{axis},i} \cup \bigcup_{i=1}^{l} \bigcup_{j=i+1}^{l} ((n_i, n_j), j - i)_{\uparrow} \cup (\text{length}, l)_{\downarrow}.
\]  

(7.28)

where \( \text{axis}_i \) is chosen from the set \( \text{child}, \text{parent}, \text{descendant}, \text{ancestor}, \text{following}, \text{following-sibling}, \text{proceeding}, \text{and proceeding-sibling}. \)

This encoding is dictated by our predicate calculus. First, the vertical position of each node, denoted by the set \( (\cdots)_{\downarrow} \), is encoded in order to evaluate predicates of type \((\uparrow P_n, \text{opt.val})\) and \((\downarrow P_n, \text{val})\). Second, the horizontal position of each node relative to its immediate left sibling, denoted by the set \( (\cdots)_{\Rightarrow} \), and its immediate right sibling, denoted by the set \( (\cdots)_{\Leftarrow} \), capture the relative position of the current node with respect to its siblings. Third, similar to the horizontal information, for each node, we also encode a node’s relative horizontal and vertical structural positioning given by the set \( (\cdots)_{\text{axis}} \). Fourth, for every pair of nodes, appearing in a single path, we compute their relative vertical position, also denoted by \( (\cdots)_{\downarrow} \), in order to evaluate predicates of type \((\uparrow P_{n_1, n_2, \text{opt.val}})\). Finally, the length of the path, denoted by \( (\text{length}, l)_{\downarrow} \), is necessary to evaluate predicates of types \((\uparrow P_{n, \text{val}})\) and \((\downarrow P_{\text{val}})\).

The set of attribute-value pairs given in Equation 7.28 constitute an event mapped from the highlighted path in the XML tree. The bound on the size of this set is formalized as follows.

**Theorem 14.** The XML encoding contains \( O(l^2) \) attribute-value pairs for each path in the XML document, where \( l \) is the length of the path and the path is assumed to have no duplicate node names.

**Proof.** It is clear from Equation 7.28, that the encoding size of vertical and horizontal positioning is linear with respect to the length of longest path in the XML document, i.e., the number of nodes in each path. However, the encoding size of relative vertical positioning is on the order of \( O(l^2) \), where \( l \) is the number of nodes in the path. Hence, the encoding size is bounded by \( O(l^2) \).
The path encoding presented in Equation 7.28 is easily generalized to path containing duplicates. The occurrence number is used to re-write the duplicate entry. For example, the path \((n_1/n_2/n_3/n_1)\) is represented as \((n_1^1/n_2^1/n_3^1/n_1^2)\). As described in [175], in order to guarantee the correctness of our XML encoding, we need to consider all different combinations of a given path with its duplicate entries removed. Therefore, if there are \(k\) duplicate node names, then we must consider \(2^k\) re-writing combinations of the original path with length \(l\), given by

\[
\bigcup_{i=1}^{k} \binom{l}{i}.
\] (7.29)

For each combination, after a set of duplicate names are removed, the occurrence numbers are adjusted accordingly. Thus, in the worst case, each path with up to \(k\) duplicates generates up to \(2^k\) subpaths, and each subpath must be mapped into a set of attribute-value pairs.

Consider the path \(p = \{n_1^1, n_2^1, n_3^1, n_4^1\}\), now after removing the duplicate node \(n_1^1\) and adjusting the subsequent sequence number, we can derive its only subpath, \(sp = \{n_2^1, n_3^1, n_1^1\}\). Without considering the subpath \(sp\), the original path \(p\) would have not matched the XPath expression \((n_2^1/n_3^1/n_1^1)\) despite the fact the the subpath \(sp\) does match the XPath expression.

We are now in position to re-state the stateless matching problem in Section 3.2.

**Definition 13.** Given a path \(p_j \in \mathcal{D}\) encoded as a set of attribute-value pairs, representing an incoming event \(\omega\), and a set of XPath expressions encoded as conjunctions of Boolean predicates, representing subscriptions \(\Omega_i\), find all subscriptions \(\Omega_i \in \Omega\) that are satisfied by \(\omega\).

---

3It is argued in [175] that duplicate node names are not common in practice; thus, not a major concern for practical purposes. For example, if a path contains only two duplicate nodes, then only two subpaths are generated; hence, the set of all generated attribute-value pairs for both subpaths are still bounded by \(O(l^2)\).

4Naturally, there is a large overlap among these subpaths that can be exploited to speed up both generation and subsequent matching computation. Especially our adaptive compressed matching algorithm introduced in Chapter 4 could be utilized.
7.4.1 XML encoding illustration

Given the XML document in Figure 7.3, in particular, focusing on the path \( p = (a^1, b^1, c^1, c^2) \), we demonstrate all possible subpaths for \( p \) and the set of attribute-value pairs necessary to represent each subpath in our formalism.\(^5\)

\[
p = (a^1, b^1, c^1, c^2) \rightarrow \\
\{ (a^1, 1)_\uparrow, (b^1, 2)_\downarrow, (c^1, 3)_\uparrow, (c^2, 4)_\uparrow \\
(a^1, 1)_\rightarrow, (b^1, 3)_\rightarrow, (c^1, 2)_\rightarrow, (c^2, 1)_\rightarrow \\
(a^1, 1)_\uparrow, (b^1, 1)_\uparrow, (c^1, 3)_\uparrow, (c^2, 1)_\uparrow \\
((a^1, b^1), 1)_{\uparrow}, ((a^1, c^1), 2)_{\downarrow}, ((a^1, c^2), 3)_{\downarrow}, ((b^1, c^1), 1)_{\downarrow}, ((b^1, c^2), 2)_{\downarrow}, ((c^1, c^2), 1)_{\downarrow} \\
(length, 4)_i \}
\]

\[
p - \{ c^1 \} = (a^1, b^1, \ast, c^1) \rightarrow \\
\{ (a^1, 1)_\uparrow, (b^1, 2)_\downarrow, (c^1, 4)_\uparrow \\
(a^1, 1)_\rightarrow, (b^1, 3)_\rightarrow, (c^1, 1)_\rightarrow \\
(a^1, 1)_\uparrow, (b^1, 1)_\uparrow, (c^1, 1)_\uparrow \\
((a^1, b^1), 1)_{\uparrow}, ((a^1, c^2), 3)_{\downarrow}, ((b^1, c^2), 1)_{\downarrow} \\
(length, 4)_i \}
\]

7.5 XML/XPath Matching

By leveraging our XML/XPath encoding, we have reduced the matching problem of XML documents against XPath expressions into the problem of matching events against conjunctions of Boolean predicates. In this section, we summarize how our Boolean predicate are evaluated given a set of derived attribute-value pairs from paths in an XML document.

**Definition 14.** To enforce the downward vertical constraint, the Boolean predicate \( P_{n, \text{opt}, \text{val}} \uparrow \) is satisfied by the event \( \omega \) such that \( \{ n, \text{val}' \}_{\uparrow} \in \omega \) and \( \text{val}', \text{opt}, \text{val} \).

**Definition 15.** To enforce the upward vertical constraint, the Boolean predicate \( P_{n, \text{opt}, \text{val}} \uparrow \) is satisfied by the event \( \omega \) such that \( \{ n, \text{val}' \}_{\uparrow} \in \omega \) and \( \{ l - \text{val}' \}_{\uparrow} \geq \text{val} \).

**Definition 16.** To enforce the forward horizontal constraint, the Boolean predicate \( P_{n, \text{opt}, \text{val}} \rightarrow \) is satisfied by the event \( \omega \) such that \( \{ n, \text{val}' \}_{\rightarrow} \in \omega \) and \( \text{val}', \text{opt}, \text{val} \).

**Definition 17.** To enforce the backward horizontal constraint, the Boolean predicate \( P_{n, \text{opt}, \text{val}} \leftarrow \) is satisfied by the event \( \omega \) such that \( \{ n, \text{val}' \}_{\leftarrow} \in \omega \) and \( \text{val}', \text{opt}, \text{val} \).

**Definition 18.** To enforce the relative vertical constraint, the Boolean predicate \( P_{n_1, n_2, \text{opt}, \text{val}} \uparrow \) is satisfied by the event \( \omega \) such that \( \{ n_1, n_2, \text{val}' \}_{\uparrow} \in \omega \) and \( \text{val}', \text{opt}, \text{val} \).

**Definition 19.** To enforce the axis predicate constraint, the Boolean predicate \( P_{n, \text{opt}, \text{val}} \) is satisfied by the event \( \omega \) such that \( \{ n, \text{count(axis)} \}_{\text{axis}} \in \omega \) and \( \text{count(axis)}_{\text{opt}}, \text{val} \).

\(^5\)In the interest of clarity, we have excluded the encoding of axes predicates, denoted by \( \langle \ldots \rangle_{\text{axis}} \), which follows similar mechanics used for expressing \( \langle \ldots \rangle_{\uparrow} \) and \( \langle \ldots \rangle_{\rightarrow} \) attribute-value pairs.
Definition 20. To enforce the path-length constraint, the Boolean predicate \( P^{\text{opt,val}} \) is satisfied by the event \( \omega \) such that \((\text{length}, \text{val'})_i \in \omega \) and \((\text{val'} \geq \text{val})\).

7.6 Summary

While most existing XML/XPath matching approaches use standard techniques such as NFAs or DFAs for processing XPath expressions, we altogether diverted and considered a fundamentally new approach for XML/XPath matching. Most importantly, we formalize and introduce a novel XPath encoding based on general value-based Boolean expressions that support horizontal and vertical navigation within an XML document and enables structural and attribute-based filtering. Furthermore, we present a dichotomy of the XPath language that can be mapped into our novel Boolean expression encoding.
Chapter 8

Data-centric Workflows-to-Kernel Mapping

8.1 Introduction

It is a common practice that business processes and workflows support globally distributed processes that involve data and participants from different geographical locations and organizations. At the same time the vast majority of workflow and business processes management (BPM) systems are either very centralized in their nature, relying on centralized processing of associated data, or support only rather restricted forms of distributed execution without considering data appropriately [23]. In such environments, for instance in global corporations, it is common that huge amounts of data need to be regularly moved across the world yet the majority of data-centric workflow systems have not focused on the distributed execution that would correspond to the actual geographical needs and similar constraints (e.g. [64]).

In recent years, there has been a growing interest in frameworks for specifying and deploying business processes that combine both data and process as first-class citizens [171, 157, 207, 7]. The shift towards data-centric business processes has a potential to address the described problem since the distribution and partitioning of a business process respecting geographical and other constraints needs to inherently focus on both process and the associated data. In this chapter, we consider one such data-centric BPM approach called Business Artifacts [157, 32, 50] and a recent meta-model for modeling Business Artifacts called Guard-Stage-Milestone (GSM) [108]. We will focus on how business processes specified in GSM can be distributed and executed on massively parallel infrastructures employing the publish/subscribe abstraction.

The ultimate goal of parallelizing a data-centric workflow is to achieve an effective distribution and grouping of flow activities and associated data fragments, respecting a set of constraints such as the infrastructure topology, geographical constraints, or pricing factors, and minimizing cost function for communication or data transport. This chapter provides the foundation for developing a mapping from data-centric workflows primitives to the publish/subscribe primitives while maintaining equivalent operational semantics. Such foundations can be applied to identify the workflow distribution that conforms to given constraints.

In the business artifacts approach, business processes are modeled as interactions of key business-
relevant conceptual entities, called Business Artifacts (or “artifacts” for short). Artifacts are modeled using an information model, that includes attributes for storing all business-relevant information about the artifact, and a lifecycle model, that represents the possible ways how the artifact might evolve. The artifact approach typically yields a high-level factoring of business processes into a handful of interacting artifact types.

The recently introduced data-centric workflow model known as Business Artifacts with Guard-Stage-Milestone Lifecycles meta-model [108] provides a substantially declarative approach for specifying artifact lifecycles that supports parallelism and modularity, with an operational semantics based on a variant of Event-Condition-Action (ECA) rules. There are four key elements in the GSM meta-model: (a) information model for business artifacts, as in all variations of the artifact paradigm; (b) milestones, which correspond to business-relevant operational objectives, and are achieved (and possibly invalidated) based on triggering events and/or conditions over the information models of active artifact instances; (c) stages, which correspond to clusters of activity intended to achieve milestones; and (d) guards, which control when stages are activated, and as with milestones are controlled through triggering events and/or conditions. Multiple stages of an artifact instance may be active at the same time, which enables the modeling of parallel activity. Hierarchical structuring of the stages supports a rich form of modularity.

The operational semantics for GSM is generalized in terms of how a single “incoming event” is incorporated into the current “snapshot” (i.e., description of all relevant aspects at a given moment of time) of a GSM system [64]. This semantics extends the well-known Event-Condition-Action (ECA) rules paradigm, and is centered around business steps (or B-steps) within data-centric workflows, which focus on the full impact of incorporating the incoming event. In particular, the focus is on what milestones (i.e., goals or objectives) are achieved or invalidated, and what stages (i.e., tasks) are opened and closed, as a result of this incoming event. Changes in milestone and/or stage status are treated as internal “status events”, and can trigger further status changes in the snapshot. Intuitively, a B-step corresponds to the smallest unit of business-relevant change that can occur to a data-centric workflow. In this chapter, we rely on the well established incremental operational semantic introduced in [64]. This semantics resembles the incremental application of the ECA-like rules, which provides an intuitive way to describe the operational semantics of a data-centric workflow (such as GSM), and provides a natural, direct approach for its implementation.

Starting with a set of data-centric workflow primitives (based on a set of acyclic ECA-style rules) and an information model that rely on an incremental operational semantics, we develop a complete mapping of data-centric workflows into the publish/subscribe abstraction. We enable this workflow transformation by redefining and formalizing key pub/sub constructs such as subscriptions and publications together with their matching condition and consumption and notifications policies. As a result, once a data-centric workflow is transformed into the pub/sub abstraction, it seamlessly inherits the distributed and loosely-coupled benefits of pub/sub. In short, we make the following contributions

- Formalizing our proposed data-centric workflow and pub/sub abstractions (Sections 8.2-8.3),
- mapping of data-centric workflows into the publish/subscribe abstraction to achieve distributed and parallel execution (Sections 8.4-8.5),
- providing a detailed theoretical analysis of our proposed mapping (Section 8.6), and
- proving the complexity of optimal workflow distribution over the publish/subscribe abstraction (Section 8.7)
8.2 Data-centric Workflow

We begin by describing a data-centric workflow known as GSM model before extracting and arguing for a set of primitives and properties required by an data-centric workflow.

8.2.1 Overview of GSM Schema

The GSM schema is defined as a set of artifact type $A$ with lifecycle having the form

$$A = (x, Att, Typ, Stg, Mst, Lcyc).$$

The essence of GSM workflow model can succinctly be described as the grouping of business processes into logical artifact type $A$ that corresponds to an actual business entity within an organization. Each artifact is comprised as a set of goal-oriented work items with lifecycles, in which a work item is modeled as stages ($Stg$) and goals are referred to as milestones ($Mst$). In addition, each artifact may have many instances ($x$) over a global shared information model in order to store their relevant data, e.g., a set of data and status attributes ($Att$) and their associated data types ($Typ$).

Furthermore, the lifecycle schema, i.e. the blueprint of how an artifact type evolves through its various stages, is provided by

$$Lcyc = (Substage, Tasks, Owns, Guards, Ach, Inv).$$

The lifecycle of each stage captures the hierarchy of its substages ($Substage$), encapsulation of a task within each (sub)stage ($Tasks$), information about stage nesting ($Owns$), conditions for enabling (sub)stages ($Guards$), conditions for determining the successful completion of (sub)stages ($Ach$), and conditions for disabling (sub)stages ($Inv$). Roughly speaking, the GSM schema defines a workflow through the lens of a stage, guards for entering a stage, and milestones for leaving a stage.

A key primitive GSM construct in addition to guard, stage, and milestone is sentry, which in fact is the building block of guards and milestones. Each sentry is a Boolean expression of type $\chi(x)$. In general, a sentry consists of two parts: the (triggering) event $\xi(x)$, which is a Boolean expression to test the type of an external event, and a condition $\varphi(x)$, which is a Boolean expression defined over the subset of GSM status attributes. A sentry takes on three different forms: (i) on $\xi(x)$ if $\varphi(x)$; (ii) on $\xi(x)$; and (iii) if $\varphi(x)$.

With the respect to GSM execution, we focus on the incremental formulation of GSM operational semantics: a variation of incremental firing of Event-Condition-Action (ECA) type rules, known as Pre-requisite-Antecedent-Consequent (PAC) rules. The set of PAC rules can be derived in polynomial time from a GSM schema [64]. More importantly, in GSM, the order of PAC rule firing is defined by the generalized notion of Polarized Dependency Graph (PDG). The PDG imposes a topological sort order on PAC rules firings, essentially a form of stratification of PAC rules, in which no cyclic relation among PAC rules are allowed which requires that PDG graph to be acyclic. The PDG imposed order on rule firing eliminates the well-known non-intuitive outcome that also arises in logic programming (in the presence of negation) and guarantees the uniqueness and the termination properties in the context of defining smallest logical unit of work, as an $B$-step, as a well-formedness of finite set of PAC rules within the $B$-step.

The incremental formulation of GSM (in turn, the execution of PAC rules in the prescribed order
Chapter 8. Data-centric Workflows-to-Kernel Mapping

of PDG) is driven and initiated upon receiving an external event from the environment. The set of all relevance PAC rules are executed in response to the external event; the firing of PAC rules are sequenced to form an atomic-step. The semantics of the B-step with respect to overall GSM system state snapshot (i.e., information model) is summarized using a 5-tuples \((\Sigma, e, t, \Sigma', Gen)\), where \(\Sigma\) is current system snapshot of the GSM instance prior consuming the ground event \(e\), \(\Sigma'\) is the new snapshot of the system after firing all relevant PAC rules that are triggered directly or indirectly by the external event \(e\), and \(Gen\) is a set of ground generated events as a result of 1-way and 2-way service calls that are encapsulated in a task; a task itself is encapsulated in a stage. Thus, the B-step is formalized with respect to the sequence firings of PAC rules such that \(\Sigma = \Sigma_0, \Sigma_1, \Sigma_2, \cdots, \Sigma_n = \Sigma'\) (where \(\Sigma_0 \neq \Sigma_1\)). Thus, after applying the \(i^{th}\) PAC rule, in order imposed by PDG, the GSM state advances from \(\Sigma_i\) to \(\Sigma_{i+1}\); in GSM, each state change is also referred to as a micro-B-step.

The key properties surrounding B-steps are that the resulting snapshot of B-step, denoted by \(\Sigma'\), is the uniqueness of \(\Sigma'\) and that the B-step always terminates. These properties are achieved in part by restricting that each \(Att\) in the GSM schema changes at most once as a result of PAC rules firing within in the context of a B-step, i.e., toggle-once principle, implying that change cannot be undone either, and in part by executing all relevant PAC rules whose consequents are reachable in the PDG graph and in order imposed by PDG, namely, visiting every reachable node in PDG using a strata-based breadth-first graph traversal.

**Definition 21.** A B-step resulting from \((\Sigma, e, t, \Sigma', Gen)\) always terminates and ends in a unique state \(\Sigma'\), where \(\Sigma \neq \Sigma'\).

The GSM schema consists of six distinct PAC rules: PAC-1 for achieving guards; PAC-2 for achieving milestones; PAC-3 for invalidating milestones; PAC-4 (or PAC-4\(^{simp}\)) for invalidating guards; PAC-5 for closing stage when milestones is achieved; and PAC-6 for closing substage when their top-level stage is closed. For complete details of PAC rules please consult [108].

The GSM execution model assumes a global external-event queue, and the current GSM operational semantics is serialized with respect to the external-event queue. In this chapter, we also rely on a global event queue to orchestrate concurrent B-steps execution such that the event queue behaves as pseudo-global clock. However, we interleave and pipeline the processing of multiple B-steps over a loosely and decoupled distributed infrastructure, in which each B-step is associated to a different external event. We achieve this distributed and parallel execution while guaranteeing an identical behavior as if B-steps were processed strictly in a sequential order and a centralized environment dictated by the global event queue.

### 8.2.2 Generalized Data-centric Workflow

Finally, we propose a generic data-centric workflow constructs and desired properties extracted from GSM schema and operational semantics. We argue that a possible abstraction for a data-centric workflows would consists of

- a set of (acyclic) rules (i.e., ECA-style rules)

- an information model for storing workflow relevant data (global workflow data schema)

\(^1\)The PAC rule’s Prerequisite is satisfied.

\(^2\)After applying the PAC rule’s Consequent, the current state of GSM instance changes.
• a well-defined notion of business-driven execution step triggered upon receiving a new external event (formalized as B-step)

• a set of desired properties such as retaining workflow history, tracing and debugging workflow, supporting analytics on workflow information model, and operating using a finite business-driven execution step\(^3\) (resulting from firing a finite set of rules in a B-step)

• a consistent view of workflow information model

We formalize our proposed data-centric workflow schema as

\[ \Gamma = (\mathcal{R}, \mathcal{I}) \]

- \( \mathcal{R} \) is essentially a set of acyclic PAC rules (or any other forms of ECA-style rules)

- \( \mathcal{I} \) is the workflow information model (or a global workflow data schema) that consists of a set of status and data attributes represented by a set of ordered pairs of \((\text{attr}, \text{value})\)

In addition, the operational semantics of \( \Gamma \) follows the the general notion of incremental operational semantics as given by Definition 21. Furthermore, to enforce acyclicity ordering of PAC rules firing w.r.t. incremental semantics, we utilize the concept of Polarized Dependency Graph (PDG) introduced in [108].

### 8.3 Publish/Subscribe Schema

In this section, we present the necessary mathematical formalization of publish/subscribe (pub/sub) for proving the correctness of our mapping from data-centric workflow to pub/sub. At the core of pub/sub abstraction lies a set of publications and subscriptions. Each publication \( P \) is defined as

\[ P = (\mathcal{E}) \]

where \( \mathcal{E} \) defines the publication’s event schema which includes the event type. The event schema consists of set of ordered pairs of \((\text{attr}, \text{value})\). An instance of publisher continuously produces events over time that conforms to the publication’s event schema.

Each subscription \( S \) is defined as follows:

\[ S = (T, x, D, \Phi(\rho_k), \delta(\rho_k), N(\rho_k)) \]

where \( \rho_k = (e, t, x) \), i.e., event type \( e \), event time \( t \), and subscription instance \( x \).

- \( T \) is the name of the subscription type, where subscription types \( T_i \) are pairwise distinct.

- \( x \) is a variable that ranges over the IDs of instances of \( S \). This is called the context variable of \( S \)

\[^3\text{All these desired properties are partly achievable by introducing the toggle-once property.}\]
Chapter 8. Data-centric Workflows-to-Kernel Mapping

- $\mathcal{D}$ is the data schema, in general, describing the internal state of each subscription $\mathcal{S}$. The unique key of the schema $\mathcal{D}$ is formed together by the triplet the type of the event $\text{eventType}$, the event time $\text{eventTime}$, and the subscription instance ID $\text{subscriptionInstance}$. Also there is a column associated to every on-event $\text{onE}_i$ appearing in PAC rules that indicates whether the event $\text{onE}_i$ was received, every data attribute $d_i$ (i.e., application and user data) and status attribute $s_j$ (i.e., internal workflow states) appearing in the logical expression of PAC rules. The tuple having $\rho_k$ as its key maintains the values for every $d_i, s_j \in \Phi$ as the result of receiving the event $\rho_k$. The final field $\text{isVisited}$ in $\mathcal{D}$ indicates whether or not all the values in this tuple has been stabilized, i.e., the stabilized value will no longer change as a result of external event $\rho_k$. Setting $\text{isVisited}$ to true in the tuple with the key $\rho_k$ implies that this tuple is now a read-only tuple and any notification (event generation) associated with $\mathcal{S}$ for event $\rho_k$ has been completed. A read-only tuple is retained for maintaining execution history and for enabling parallel and distributed processing of PAC rules. Therefore, tuples in $\mathcal{D}$ are uniquely identified by the triple $\rho_k$.

$$\mathcal{D}(\text{eventType}, \text{eventTime}, \text{subscriptionInstance},$$
$$\text{onE}_1, \cdots, \text{onE}_n, d_1, \cdots, d_n, s_1, \cdots, s_p, \text{isVisited})$$

We extend each domain type (for on-event, data and status attributes) with the special symbol $\emptyset$, indicating that the current value is unstable. In addition, we consider the original domain values (not in the extended set) as stabilized.

- $\Phi(\rho_k)$ is subscription matching condition, where $\Phi(\rho_k)$ is a disjunction over $\phi_i(\rho_k) \in \Phi(\rho_k)$, where each $\phi_i(\rho_k)$ is a condition, a logical formula, that is instantiated and correlated with each external event $\rho_k$. This condition is expressed over the condition language $\mathcal{L}$ that is a subset of First-Order Logic (FOL) that supports i) scalars values, ii) binary relations: logical operators ($\lor, \land, \rightarrow, \forall$), where $\forall$ is an exclusive $\lor$, and arithmetic operators $(<, \leq, =, \neq, \geq, >)$, iii) unary relation ($\neg$), iv) quantifications over instances of a subscription ($\forall$ and $\exists$), v) functions:

$$\begin{align*}
- & \tau_k(\text{attr}, \rho_k), \text{or simply, } \tau(\text{attr}), \text{returns the current value of the attribute } \text{attr} \text{ w.r.t. } \rho_k. \\
- & \tau_{k-1}(\text{attr}, \rho_k) \text{ returns the last value of the attribute } \text{attr} \text{ w.r.t. } \rho_k \text{ for } k > 2; \text{ otherwise returns False for Boolean attributes, and a default or a null (⊥) value for non-Boolean attributes.}^4
\end{align*}$$

Finally, we resort to three-valued logic, where we have three possible values ($\text{True, False, Unknown}$), where Unknown is the interpretation of the unstable value represented by $\emptyset$ (we do not consider null value (⊥) as unstable and we do not permit null value for Boolean variables). We define the evaluation of any logical binary or unary operator involving an Unknown as Unknown with the exception of $\forall$ (for which as long as one of the operands is True the expression will evaluate to True as well); whereas we rely on traditional two-valued logic when no Unknown is present. Also when dealing with different system snapshots $\Sigma$, to differentiate an attribute value among different snapshots when not clear from the context, we extended the definition $\tau$ to include $\Sigma$ as input parameter as follows $\tau_k(\Sigma, \text{attr}, \rho_k)$.

- $\delta$ is the subscription’s consumption policy that describes how the internal state of subscription changes after consuming an event.

^4In fact, the function $\tau_{k-1}$ is only used for $S_i$ where by the definition, each $S_i$ is initially set to False.
• \( N(\rho_k) \) is the subscription’s notification policy that is also a disjunction over \( \nu_i(\rho_k) \in N(\rho_k) \) and instantiated and correlated with each external event \( \rho_k \). The notification consists of notification schema that describes the content of the event (its payload), and a set of condition \( \nu_i(\rho_k) \) that dictates how the content of the event is generated.

Furthermore, we define the relationship between subscription’s condition \( \Phi \) and subscription’s notification policy \( N \) by the relation (as an ordered pair), where each \( \phi_i \in \Phi \) is associated to the corresponding \( \nu_i \in N \), meaning, when the matching condition \( \phi_i \) is satisfied, then notification condition \( \nu_i \) is evaluated.

\[
\Psi_s(\rho_k) = \bigcup_{\phi_i \in \Phi} (\phi_i(\rho_k), \nu_i(\rho_k)).
\]

An instance of the subscription \( S \) consists of an internal state \( \Sigma_j^S \) over the subscriptions’s schema \( D \). The internal state of subscription can only be changed upon receiving (consuming) an external event or generating an event (notification). In general, the internal state together with an external event shapes the subscription operational semantics \( O_S \) (a.k.a. the matching semantics), which is summarized as a 6-tuples

\[
O_S = (\Sigma_j^S, e, t, x, \Sigma_{j+1}^S, Gen).
\]

• \( \Sigma_j^S \) is the current internal state of the subscription \( S_j \); the internal state of a subscription represents states of all its instances.

• \( e \) is a ground occurrence of an external event.

• \( t \) is the logical time which is greater than all logical timestamps occurring in \( \Sigma_j^S \).

• \( x \) is a variable that ranges over the IDs of instances of \( S \). This is all referred to as the context variable of \( S \).

• \( \Sigma_{j+1}^S \) is the internal state after consuming event \( e \).

• \( Gen \) is the set of ground generated event occurrences.

Consequently, the subscription operational semantics \( O_S \) is formally expressed as follows.

**Definition 22.** Given a subscription \( S \) with the current internal state \( \Sigma_j^S \) and an external event \( e \) at time \( t \) for the instance \( x \) (denoted by \( \rho \)) of \( \Sigma_j^S \), the subscription \( S \) examines \( e \) and either accepts \( e \), transitions from \( \Sigma_j^S \xrightarrow{e, t, x} \Sigma_{j+1}^S \) or rejects (or drops) \( e \).

We are now in position to formally define the publish/subscribe schema \( \Pi \) as

\[
\Pi = (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mathcal{C})
\]

• \( \mathcal{P} \) is a set of publications.

• \( \mathcal{S} \) is a set of subscriptions.

• \( \mathcal{E} \) is the global event schema that captures both publication’s event schema and subscription’s notification schema. The global event schema consists of set of ordered pairs of (\( attr, value \)).
Chapter 8. Data-centric Workflows-to-Kernel Mapping

165

- \( \mathcal{C} \) is the communication state schema such that for each external event \( e \) maintains the event type of \( e \), the logical time of event \( e \), the subscription type that has processed event \( e \), the subscription instance ID that has processed event \( e \), summarized as

\[
\mathcal{C}(\text{eventType}, \text{eventTime}, \text{subscriptionType}, \text{subscriptionID})
\]

Publish/Subscribe Operational Semantics

In order to discuss and construct a mathematical model of our publish/subscribe abstraction, we make the following assumptions—this assumptions are required to prove the correctness of our mapping (cf. Section 8.6) only when there are more than one external event publisher:

1. Each subscription instantaneously examines a single external event according to subscription operational semantics.
2. In any instant of time, only a single subscription is examining an external event in \( \Pi \).

An instance \( I \) of \( \Pi \) is defined as a sequence of global snapshot of \( \Sigma_1 \cdots \Sigma_k \) over a discrete time space \( t \), where \( \Sigma_i = \{ \Sigma^C_k, \Sigma^S_j \} \), \( \Sigma^C_k \) is the communication state at time \( t \) over \( \Pi \), \( \Sigma^S_j \) is an internal state of each instance of subscription \( S \). The \( \Pi \) operational semantics is summarized as

\[
\mathcal{O}_\Pi = (\Sigma_k, e, t, x, \Sigma_{k+1}).
\]

- \( \Sigma_k \) is the current global snapshot of \( \Pi \).
- \( e \) is a ground occurrence of an external event that is pending, implying that there is at least one subscription instance \( x \) that has not examined \( e \) yet.
- \( t \) is the logical time of event \( e \).
- \( x \) is a variable that ranges over the IDs of instances of \( S \).
- \( \Sigma_{k+1} \) is the new global snapshot of \( \Pi \)

The \( \Pi \) operational semantics is formally expressed as follows.

**Definition 23.** Given a pending event \( e \) with the logical time \( t \) and the instance \( x \) of the subscription \( S \) that has yet to examine \( e \), with \( S \) having the current state \( \Sigma^S_j \), then the global snapshot advances instantaneously from \( \Sigma_k \xrightarrow{e,t,x} \Sigma_{k+1} \), namely,

1. the communication state transitions from \( \Sigma^C_k \xrightarrow{e,t,x} \Sigma^C_{k+1} \) signifying that the event \( e \) was sent to \( S \) for instance \( x \)
2. the subscription \( S \) examines the external event \( e \) in accordance to \( \mathcal{O}_S \); hence, \( S \) either accepts \( e \) and transitions from \( \Sigma^S_j \xrightarrow{e,t,x} \Sigma^S_{j+1} \) or rejects \( e \).

We define a valid execution sequence over \( \Pi \) as one that corresponds to pseudo-serializable execution such that in any instant of time, \( \Pi \) transitions only once from state \( \Sigma^C_k \xrightarrow{e,t,x} \Sigma^C_{k+1} \) and only a single instance of subscription \( S \) receives an event \( e \) and transitions from \( \Sigma^S_j \xrightarrow{e,t,x} \Sigma^S_{j+1} \) (if necessary).
Notably, in any time instance, many subscriptions (or many instances of a single subscription) may be waiting to receive the event $e$; however, \textit{pseudo-serializable execution property} does not impose any restriction on the order which subscriptions (or instances of a subscription) must receive the event $e$. Therefore, any non-deterministic selection of subscriptions (or instances) suffices that results in a instantaneous examination of event $e$ at time $t$ by a single subscription instance $x$. Most importantly, this pseudo-serialization requirement can be dropped when there is a single publisher of external events (a plausible assumption that also holds in the GSM semantics).

At the global level of $\Pi$, an event is pending only if at least one subscription instance has not examined it yet, and (in theory) every subscription instance must examine every event exactly once. Therefore, from the communication state $C$, it can be inferred, which events have been processed for which instances of subscription $S$ and which events are pending for which instances of $S$.

Finally, in general, with more than one publisher of external events (not yet deemed important in GSM semantics) any valid implementation of $\Pi$ must guarantee pseudo-serializable execution property.

\section{Workflow Mapping Overview}

Given a schema $\Gamma = (\mathcal{R}, \mathcal{T})$, we construct a pub/sub schema $\Pi = (\mathcal{P}, \mathcal{S}, \mathcal{E}, C)$ such that $M : \Gamma \rightarrow \Pi$. The set $\mathcal{P}$, in our mapping consists of single publisher which simply publishes the external events coming from the environment. However, constructing the set of necessary subscriptions is more subtle and is primarily derived from the set of the PAC rules and PDG formulation in a given schema $\Gamma$. In addition, we are required a set of subscriptions for bookkeeping purposes such as updating data and status attributes and determining start and end of a $B$-step.

Consequently, we define subscriptions both for processing relevant PAC rules and maintaining the current values for status attributes (e.g., guards, stages, and milestones). In general, two key classes of subscription conditions arises: (1) the \textit{application-specific condition} which captures core of $\Gamma$ operational semantics that encodes both the PAC rules semantics and PDG topological sort order semantics and (2) the \textit{generic condition} which implements a bookkeeping mechanism such as providing a consistent view of the data (i.e., status attributes) with an implicit locking strategy. These two classes of conditions also incorporate the time semantics of $\Gamma$ schema which is based on the external event received from the single publisher in our $\Pi$ formulation. Therefore, subscriptions are event-relativized in a sense that each subscription evaluates its conditions, sends its notification, and implements its consumption policy.
(maintaining its internal state) in the context of each external events in isolation, which forms a B-step.

In our mapping \( M : \Gamma \rightarrow \Pi \), we are required the following set of subscriptions for key \( \Gamma \) operations: \([S_{⊕s}]\) and \([S_{⊖s}]\) for satisfying/falsifying or validating/invalidating status attributes \( s \); \([S_s]\) for updating the status attribute \( R.s \); \([S_d]\) for updating the data attribute \( R.d \); \([S_{source}]\) for identifying the beginning a B-step; and \([S_{sink}]\) for identifying the of ending of a B-step, where \( ⊕ \) or \( ⊖ \) polarity, denotes a positive or a negative change in status attributes, namely, a achieving and invalidating a sentry, respectively.

Next, we provide a high-level overview of each subscription. The high-level representation and interaction among subscriptions (represented as oval) is also depicted in Figure 8.1; intuitively, the directed solid arrows indicate the flows of events among subscriptions and the (bright-colored) directed dashed arrows indicate events received from and sent to the environment while the (black) dashed line are bookkeeping messages for maintaining consistent view of attributes. What is not shown in the figure, for improved readability, is that there must be an arrow from every node to the node \( S_{sink} \).

Furthermore, the precise meaning of the arrows become evident in Section 8.5, after formally defining each subscription.

\([S_{⊕s}], [S_{⊖s}]\) For each status attribute \( s \) in \( \Gamma \) information model \( I \), we add the subscription \( S_{⊕s} \), for validating the attribute \( s \), and the subscription \( S_{⊖s} \), for invalidating the attribute \( s \). The subscription’s condition \( Φ \) is derived based on PAC rules prerequisite and antecedent conditions. Hence, \( Φ \) is an application-specific condition.

\([S_s]\) For each status attribute \( s \) in the information model \( I \), we add the subscription \( S_s \) that listens to updates (i.e., notifications of \( S_{⊕s} \) and \( S_{⊖s} \)) for value \( s \). Hence, \( S_s \)’s \( Φ \) is a generic condition.

\([S_d]\) For each data attribute \( d \) in the information model \( I \), we add the subscription \( S_d \) that listens to updates on \( d \) at the outset of the B-step. Hence, \( S_d \)’s \( Φ \) is also a generic condition.

\([S_{source}, S_{sink}]\) For identifying the beginning and ending of the B-step we add the source subscription \( S_{source} \) and the sink subscription \( S_{sink} \), respectively. All of these subscriptions are intended for bookkeeping purposes; thus, their subscription’s conditions are also generic.

8.5 Mapping Formalization

The subscription plays the central role in formulation of our data-centric workflows \( \Gamma = (R, I) \) into the our publish/subscribe abstraction given by \( Π = (P, S, E, C) \). We formalize the subscription semantics as follows:

\[
S = (T, x, D, Φ(ρ_k), δ(ρ_k), N(ρ_k)),
\]

in which its condition \( Φ(ρ_k) \), consumption policy \( δ(ρ_k) \), and notification policy \( N(ρ_k) \) are instantiated and correlated with \( ρ_k \). In addition, we define the relation \( Ψ \) to capture the association between a matching condition and its corresponding notification policy within a subscription, denoted by

\[
Ψ_{i,s}(ρ_k) = (φ_i(ρ_k), ν_i(ρ_k)).
\]

8.5.1 Matching and Notification Policies

In this section, we start by providing a detail account of our novel mapping of \( \Gamma \) application-specific semantics, namely, encoding of PAC rules and PDG topological sort order, into a set of subscriptions in a purely publish/subscribe setting. In addition, we provide the foundation for emulating \( \Gamma \) generic-
execution semantics including the necessary bookkeeping mechanism in order to guarantee \( \Gamma \) mapping correctness by ensuring data consistency and the \( B \)-step semantics, which is an essential abstraction of \( \Gamma \) semantics. Likewise, \( \Gamma \) generic-execution semantics are also emulated through a novel mapping into a set of subscriptions.

**PAC Rules and PDG Mapping**

We first define the \( \Gamma \) application-specific conditions for \( S \), which is also the key component of the relation \( \Psi_{i,s}(\rho_k) \). Each logical formula \( \phi_i(\rho_k) \in \Phi \) is defined as

\[
\phi_i(\rho_k) = \psi_{i,PDG}(\rho_k) \land \psi_{i,PseudoClock}(\rho_k)
\]

where

- \( \psi_{i,PDG} \) is the PDG predecessors component, that is, a logical formula that encodes the PDG topological sort order semantics, i.e., \( \psi_{i,PDG} \) is a logical formula that evaluates to true when all variables in \( S \) have been stabilized.

- \( \psi_{i,PseudoClock} \) is a logical formula that enforces that subscriptions are processed based on the order of external events, i.e., event-order serialization.

The second component of \( \Psi \), that is, the notification expression \( \nu_i(\rho_k) \in N \) follows

\[
\nu_i(\rho_k) = \begin{cases} 
\gamma_{\rho_k}, S_{\rho_k}^{visited} & \text{if } \psi_{i,SAT}(\rho_k) \\
\neg \gamma_{\rho_k}, S_{\rho_k}^{visited} & \text{if } \forall \nu_i \in N, \neg (\psi_{i,SAT}(\rho_k)) \\
\text{WAIT} & \text{if } \exists \phi_i \in \Psi_i, \neg (\phi_i)
\end{cases}
\]

- \( S_{\rho_k}^{visited} \) is an event that indicates that the subscription \( S \) for the event of type \( T \) was successfully visited, i.e., (partially) completed as defined in section 8.5.2.

- \( \gamma \) is the consequent of the PAC rule indicating a change (either a positive or a negative) to a particular status attribute \( s \) in \( \Gamma \) information model (\( \gamma = \odot s \)), while \( \neg \gamma \) indicates no change to the status attribute \( s \). In addition, each of \( \gamma_{\rho_k} \) and \( \neg \gamma_{\rho_k} \) is an event that reports the current value of the status attribute \( s \) in the context of the external event \( \rho_k \).

- \( \psi_{i,SAT}(\rho_k) \) is a logical formula derived from PAC rule’s prerequisite \( \pi \) and antecedent \( \alpha \).

- \( \text{WAIT} \) is an indicator that implies not all subscription’s condition \( (\phi_i) \) have been satisfied.

To construct the application-specific condition, we adapt the PDG construction algorithm [108], which operates based on a given set of \( \Gamma_{PAC} \). Suppose each PAC rule has the form \((\pi, \alpha, \gamma)\) that stands for prerequisite, antecedent, and consequent, respectively.

The antecedent \( \alpha_i \) of the PAC rule is derived from the form \textbf{on} \( \xi(x) \) \textbf{if} \( \varphi(x) \), where each expression \( expr \in \xi(x) \) is of the form on-event \( \text{onEventType} \) or \( \odot s \), where \( \text{onEventType} \) indicates waiting for an external event of the type given by \( \text{onEventType} \), while \( \odot \) means waiting for positive \( \oplus \) or negative \( \ominus \) change in status attribute. Similarly, every expression \( expr \in \gamma \) also follows the form \( \odot s \). However, expression \( expr \in \varphi(x) \) has the form \( s \), which simply indicates an stable value for status attribute \( s \) in the artifact type \( R \), a value is stabilized if it will no longer be changed in the current \( B \)-step.
We first collapse instances of PAC rules \( R_i \in \mathcal{R} \) that have identical \( \pi \) and \( \gamma \) into a super PAC rule given by \((\pi, A, \gamma)\), where \( A = (\forall \alpha \in R_i, \alpha) \). In general, PAC rules share identical \( \pi \) and \( \gamma \) because for a given status attribute \( s \), there may exist multiple rules for satisfying and falsifying it. The notion of super PAC rule further simplifies the mapping formulation.

Therefore, the (super) PAC rule \( R \) is be mapped to the \( S \circ s \), where \( \circ s \in \gamma \). The relation \( \Psi \circ s \in S \circ s \) (an application-specific condition and notification) is constructed through a various stages of mapping, which are described next.

Therefore, each PAC rule is used to construct the subscription condition: \((\pi, A, \gamma) \in R \rightarrow \Phi \in S \circ s\). More precisely, we can derive each \( \phi_i \in \Phi \) based on the PAC rule as follows:

\[
\mathcal{M}_\Phi: (\alpha_i \in A) \in R \rightarrow \phi_i \in \Phi.
\]

The key component of \( \phi_i \), denoted by \( \psi_{i,PDG} \in \phi_i \), is at the core of subscription mapping that incorporates the notion of PDG predecessors, an integral part of encoding the PDG topological sort order semantics of \( \Gamma \) into \( \Pi \). This stage of mapping is represented by

\[
\mathcal{M}_{\psi_{i,PDG}}: \alpha_i \in R \rightarrow \psi_{i,PDG} \in \phi_i.
\]

Thus, for each \( \phi_i \in \Phi \), we construct \( \psi_{i,PDG} \in \phi_i \) from the corresponding \( \alpha_i \in R \). The actual definition of \( \psi_{i,PDG} \) is derived by adapting the PDG construction algorithm by examining the antecedent component of each PAC rule that coarsely identifies the set of status attributes whose values must be stabilized before firing a PAC rule, i.e., evaluating a subscription. We formally, define \( \psi_{i,PDG} \) as a set of on-event that listens for positive or negative changes of variables appearing in the PAC rule’s antecedent.

\[
\psi_{i,PDG}(\rho_k) = \left( \bigwedge_{\circ s \in \xi(x)} \tau_k(\text{on}\circ s, \rho_k) \lor \tau_k(\text{on}\circ \overline{s}, \rho_k) \right) \land \\
\left( \bigwedge_{s \in \phi(x)} \tau_k(\text{on}\cdot s, \rho_k) \right),
\]

(8.5)

where \( \text{on}\circ s \) and \( \text{on}\circ \overline{s} \) refer to events that announces a change or no change to \( \circ s \) while \( \text{on}\cdot s \) refers to an event that holds the current value of \( s \).

The second component of \( \phi_i \) is \( \psi_{PseudoClock} \) which enforces that each subscription is processed, namely, it is condition \( \phi_i \) is satisfied, in order in which external events are arrived. Therefore, the external events acts as a pseudo clock; the enforcement of pseudo clock is defined as logical formula as follows

\[
\psi_{PseudoClock}(\rho_k) = (\# \rho_j, \rho_j \in \Sigma^S, \neg (\tau_j(\text{isVisited}, \rho_j)) \land \\
\tau_j(\text{eventTime}, \rho_j) < \tau_k(\text{eventTime}, \rho_k) \land \\
\tau_j(\text{subscriptionInstance}, \rho_j) = \tau_k(\text{subscriptionInstance}, \rho_k)).
\]

(8.6)

Therefore, for each subscription instance once the PDG requirement \((\psi_{i,PDG} \in \phi_i)\) is satisfied, namely, all variables in \( \alpha \) has been stabilized, and all prior external events have been processed \((\psi_{PseudoClock} \in \phi_i)\), then the corresponding notification component of \((\phi_i, \nu_i)\) is triggered. Each \( \nu_i \in N \)
is partially derived from the corresponding component of the super PAC rule $(\pi, \alpha_i, \gamma)$ in accordance to Equation 8.4.

$$M_N : (\pi, \alpha_i, \gamma) \in R \rightarrow \nu_i \in N.$$  

In particular, the key component of $\nu_i$ is a logical formula $\psi_{i,SAT}$, which capture the behavior of the notification policy. Before, giving the definition of the logical formula $\psi_{i,SAT}$, we must re-write its key component: $\pi$ and $\alpha_i$. This re-writing is necessary for abiding by the $\Gamma$ semantics, in which each variable in $\pi$ must use its last recent value from the last completed B-step (if any) while each variable in $\alpha_i$ must use its most recent value.

Thus, we re-write $\pi$ (whose consists of only Boolean variables) as follows

$$M_\pi : s_i \in \pi \rightarrow \tau_{k-1}(s_i, \rho_k),$$

Similarly, we re-write the $\alpha$ (whose consists of both status and data attributes) based on the most recent values as follows ($M_{\alpha_i}$ is consists of three stages of re-writing given by $M_{\phi \in \alpha_i}$, $M_{\xi \in \alpha_i}^1$, and $M_{\xi \in \alpha_i}^2$)

$$M_{\phi \in \alpha_i} : a_i \in \phi \rightarrow \tau_k(a_i, \rho_k),$$

$$M_{\xi \in \alpha_i}^1 : \circ s \in \xi \rightarrow \tau_k(\circ s, \rho_k),$$

$$M_{\xi \in \alpha_i}^2 : onEventType \in \xi \rightarrow \tau_k(onEventType, \rho_k) = onEventType.$$

Finally, the mapping of PAC rule to $\psi_{i,SAT}$ is expressed as

$$M_{\psi_{i,SAT}} : (\pi, \alpha_i) \in R \rightarrow \psi_{i,SAT} \in \phi_i,$$

where $\psi_{i,SAT}$ is simply derived by conjunction of re-written $\pi$ and $\alpha$

$$\psi_{i,SAT}(\rho_k) = M_\pi \land M_{\alpha_i}. \quad (8.7)$$

**Data Consistency & Semantics Simulation**

Now that we have successfully demonstrated our novel mapping to transform the PAC rules into as set of subscriptions, next, we show the necessary subscriptions for general bookkeeping and overall execution of the $\Gamma$ based on subscription generic condition defined next.

Given the relation $\Psi_{i,s}(\rho_k) = (\phi_i(\rho_k), \nu_i(\rho_k))$, then the generic condition $\phi_i$ is defined by

$$\phi_i(\rho_k) = \tau_k(\circ a, \rho_k) \lor \tau_k(\circ \overline{a}, \rho_k), \quad (8.8)$$

where $\phi_i$ essentially captures the interest in any attempt to alter the value of attribute $a$: a change ($\circ a$) or no change ($\circ \overline{a}$). On the other hand, the notification policy $\nu_i$ is expresses as

$$\nu_i = \underbrace{\tau_k(a, \rho_k)}_{\text{event-relativized status update}} \land S_{\rho_k}^{visited}, \quad (8.9)$$

where $\nu_i$ is responsible to broadcast the current value of $a$.

We start with $\Gamma$ data consistency requirement that ensures that a consistent view of status attributes;
thereby, we must ensure that when a status attribute changes, no race condition for updating the value is
arised and that every interested subscription has the most up-to-date values of its status attributes. To
achieve data consistency, we add the subscription $S_s$ with a generic condition for every status attribute
$s$ in the $\Gamma$ information model, which acts as a single gateway for changing $s$ value and subsequently
broadcasting the final stabilized value of $s$ to all interested subscriptions. The relation $\Psi_s \in S_s$ is given
by:

$$
\phi_s(\rho_k) = \tau_k(\text{on} \oplus s, \rho_k) \lor \tau_k(\text{on} \ominus s, \rho_k) \lor
(\tau_k(\text{on} \permute s, \rho_k) \land \tau_k(\text{on} \circledast s, \rho_k))
$$

$$
\nu_s(\rho_k) = \begin{cases} 
\tau_k(s, \rho_k) \leftarrow \text{True}, S_{\text{visited}}^{\text{source}} & \text{if } \tau_k(\text{on} \oplus s, \rho_k) \\
\tau_k(s, \rho_k) \leftarrow \text{False}, S_{\text{visited}}^{\text{source}} & \text{if } \tau_k(\text{on} \ominus s, \rho_k) \\
\tau_k(s, \rho_k) \leftarrow \tau_{k-1}(s, \rho_k), S_{\text{visited}}^{\text{source}} & \text{otherwise},
\end{cases}
$$

(8.10)

where $\leftarrow$ indicates assignment of the value of the right-side to the variable on the left-side.

Likewise, we construct a set of subscriptions which listens to events containing value for each data
attribute. Upon consuming an external event, if the value in the event payload is different from current
value, then the subscription $\Psi_d$ generates the value, derived from the change or no change events,
accordingly, as follows:

$$
\phi_d(\rho_k) = \tau_k(\text{on} \Delta e_{\rho_k}, \rho_k)
$$

$$
\nu_d(\rho_k) = \begin{cases} 
\ominus d_{\rho_k}, \tau_k(d, \rho_k), S_{\text{visited}}^{\text{source}} & \text{if } d \in \Delta_{\text{source}} \land \\
\ominus d_{\rho_k}, \tau_k(d, \rho_k), S_{\text{visited}}^{\text{source}} & \text{if } d \in \Delta_{\text{source}} \land \tau_k(d, \rho_k) \\
S_{\text{visited}}^{\text{source}}, \tau_k(d, \rho_k) \leftarrow \tau_{k-1}(d, \rho_k) & \text{otherwise},
\end{cases}
$$

(8.11)

where $\Delta_{\text{source}}$ summarizes data attributes appearing in $e$.

Lastly, in $\Gamma$ simulation, it is crucial to identity the beginning and ending of a completed B-step.
Therefore, first, we focus on the outset of a new B-step, which is achieved through subscription $S_{\text{source}}$.
The source subscription $S_{\text{source}}$ has a special property because it is the only subscription that waits upon
only receiving the external event $e$ from environment, which in turn establishes the beginning of a new B-step upon receiving the new event (B-step deterministic-initiation property). Therefore, $S_{\text{source}}$ sends the events $S_{\text{visited}}$ and $e_{\rho_k}$ that is understood by all subscriptions\(^5\), where its type, time, and intended subscription instances are summarized in $\rho_k$; hence the relation $\Psi_{\text{source}} = (\phi(\rho_k), \nu(\rho_k))$ is given as

\[
\phi_{\text{source}}(\rho_k) = e \\
\nu_{\text{source}}(\rho_k) = S_{\text{visited}}, e_{\rho_k}, \Delta e_{\rho_k}.
\]

In order to guarantee B-step deterministic-initiation property, we must add on $S_{\text{visited}}$ to all subscriptions whose $\phi_i \in \Phi$ are empty. In the same spirit, the end of B-step is determined by introducing $S_{\text{sink}}$ that subscribes to every subscriptions involved in order to establishes the ending of a B-step (B-step deterministic-completion property). Hence, $\Psi_{\text{sink}}(\rho_k)$ is given by

\[
\phi_{\text{sink}}(\rho_k) = \bigwedge_{S_i \in S'} \tau_k(\text{on} S_{\text{visited}}^{\rho_k}, \rho_k) \\
\nu_{\text{sink}}(\rho_k) = S_{\text{visited}},
\]

where $S' = S \setminus S_{\text{sink}}$

### 8.5.2 Consumption Policy

One of the key aspect of consumption policy is how to update the internal state of each subscription $\Sigma$. The consumption policy is tightly tied to the subscription operational semantics, denoted by $O_S = (\Sigma_S, e, t, x, \Sigma_{S+1}, \text{Gen})$, in which the subscription $S$ receives a single event $e$ at time $t$ for subscription instance $x$, and $S$ either accepts $e$ and advances its internal state or drops $e$.

In order to precisely model consumption policy w.r.t. to subscription operational semantics, we need to discuss the subscription evolution as it goes through various lifecycle within a B-step: initiation, modification, completion, satisfaction, generation, and termination. Each stage and its interaction with other stages is defined next and also illustrated in Figure 8.2.

**Stage 1.** The subscription initiation occurs for the event associated to $\rho_k$ (within the $k$th B-step), when the subscription is first receive either directly (the event $e_{\rho_k}$) or indirectly (such as status or data attributes update in the context of $\rho_k$), then eventType, eventTime, and subscriptionInstance are populated based on $\rho_k$ and isVisited is set to false while the rest of its attributes in $D$ are set to $\emptyset$. However, if the subscription instance $x \in \rho_k$ does not exist in $\Sigma_S$, then as part of the initialization (and creation of new instance), all status attributes are set to false and all data attributes are set to their default values (possibly the null values).

**Stage 2.** The subscription modification occurs for the event associated to $\rho_k$ (within the $k$th B-step) after the subscription has been initiated (before or after of subscription partial completion), when the internal states of the subscription is updated and it is transitioned according to the subscription operational semantics:

\[
O_S = \Sigma_S \xrightarrow{E_{\rho_k} \rightarrow (e,t,x)} \Sigma_{S+1}.
\]

\(^5\)In essence, the content of $e$ and $e_{\rho_k}$ are identical, and the sole purpose of $e_{\rho_k}$ is to enforce that $S_{\text{source}}$ is the initiator of all events.
The internal state of the subscription changes by at most one single attribute in \( D \); hence, only one of the following changes is permissible:

1. \((\exists \text{onE}_k \in D) \rightarrow \tau_k(\text{onE}, \rho_k) \leftarrow \text{True}\)

2. \((\forall (a_i, \text{value}) \in E_{\rho_k}, a_i \in D) \rightarrow \tau_k(a_i, \rho_k) \leftarrow \text{value}\)

**Stage 3.** The subscription completion occurs for the event associated to \( \rho_k \) (within the \( k \)-th B-step) after the subscription has been initiated, when at least one of the subscription’s \( \phi_i(\rho_k) \in \Psi(\rho_k) \) is evaluated to true. If all \( \phi_i(\rho_k) \) evaluates to true, then the subscription is considered completed while if at least one of \( \phi_i(\rho_k) \) evaluated to true, then the subscription is considered partially completed.

**Stage 4.** The subscription satisfaction occurs for the event associated to \( \rho_k \) (within the \( k \)-th B-step) after the subscription has been (partially) completed, when \( \phi_i(\rho_k) \in \Psi(\rho_k) \) is evaluated to true, i.e., the subscription is (partially) completed, and subscription’s corresponding \( \nu_i(\rho_k) \) is also evaluated to true.

**Stage 5.** The subscription generation occurs for the event associated to \( \rho_k \) (within the \( k \)-th B-step) after the subscription is satisfied, when subscription’s the relevant events are generated according to the definition of \( \nu_i(\rho_k) \).

**Stage 6.** A subscription termination occurs for the event \( \rho_k \) (within the \( k \)-th B-step) after the events has been generated by the subscription, when attribute \( \tau_k(\text{isVisited}, \rho_k) \) is assigned to true. Once the \( \text{isVisited} \) is set to true, then the tuple associated to \( \rho_k \) becomes read-only.

### 8.6 Workflow Mapping Analysis

In this section, we prove that the data-centric workflow schema \( \Gamma \) under incremental formulation (sequential execution) is equivalent to pub/sub schema \( \Pi \) (distributed execution), expressed as \( M : \Gamma \rightarrow \Pi \). Before establishing the correctness and equivalence of the \( \Gamma \) and the \( \Pi \) schemas, we define a set of preliminary concepts.

As described in Section 8.2, the incremental operational semantics of \( \Gamma \) is defined as the 5-tuples \((\Sigma, e, t, \Sigma', \text{Gen})\), and the \( \Gamma \) system snapshot transition, denoted by \( \Sigma \xrightarrow{e} \Sigma' \), is defined as the smallest logical business step (B-step), which consists of sequence firings of PAC rules; the B-step in its expanded form is given by

\[
\Sigma = \Sigma_0, \Sigma_1, \Sigma_2, \ldots, \Sigma_n = \Sigma',
\]

where \( \Sigma_0 \neq \Sigma_1 \) (due to updating data attributes based on the external incoming event \( e \)) and each \( \Sigma_i \) is referred to as a micro-B-step. Thus, the \( i \)-th micro-B-step corresponds to firing of the \( i \)-th PAC rule. Furthermore, based on the \( \Gamma \) semantics, each PAC rule firing results in change of exactly one status attribute and the value of each status attribute changes at most once within a B-step (the toggle-once principle); consequently, also each PAC rule is fired at most once within in B-step. We formalize the changes in value status attribute as follows.

**Definition 24.** An status attribute is in stable state during a B-step as soon as either its value is changed due to PAC rule firing or the B-step is completed, namely, all the relevant PAC rules for external event \( e \) have been executed. Therefore, the value of stable status attribute is no longer changed as B-step reaches its completion.
Definition 25. We refer to initial and final system snapshot of B-step as complete system snapshot, denoted by $\Sigma$ (or $\Sigma_0$) and $\Sigma'$ (or $\Sigma_n$), in which all status attributes are stable.

Definition 26. We refer to intermediate system snapshot within a B-step as partial system snapshot, denoted by $\Sigma_i$, where $0 < i < n$, in which not all status attributes are stable.

Finally, we emphasize that the incremental formulation follows a sequential and central execution, in which the $\Gamma$ semantics of the B-step execution is defined as an atomic step, and each B-step consists of finite number of micro-B-steps. Therefore, we define the concept of time in terms of B-step such that system time advances only from $t_i$ to $t_{i+1}$ after processing the $i^{th}$ event ($e_i$), i.e., completion of $i^{th}$ B-step. In addition, the external events are processed in order in which they arrive—the in-order processing of external events.

Theorem 1. The $\Gamma$ incremental semantics guarantees the in-order processing of external events (when all events are published from a single source). Hence, the B-step execution (i.e., PAC rules firing) follows the event-order serialization.

Proof. The proof is simply follows from the $\Gamma$ incremental semantics such that external events are consumed in-order and each consumed event (potentially) triggers a B-step that is executed atomically.

Similar to the B-step event-order serialization in the $\Gamma$ semantics, the micro-B-steps within a B-step also follow a strict order which is imposed by topological sort order of the PDG—the PDG-based serialization of micro-B-steps.

Theorem 2. The $\Gamma$ incremental semantics guarantees the PDG-based serialization of micro-B-steps.

Next, we show how the operational semantics of $\Gamma$ is also guaranteed in our pub/sub formulation. As provided in Section 8.3, our pub/sub schema $\Pi$ operational semantics is also formalized as a sequence of changes in (global) system snapshot denoted as $\Sigma_i \rightarrow e.t.x \rightarrow \Sigma_{i+1}$, implying a single subscriber received and accepted event $e$.

Lemma 3. Our pub/sub operational semantics guarantees two basic properties: (1) in-order delivery of events between any pair of publisher and subscriber and (2) no message lost.

Proof. These properties are direct consequences of our pub/sub definition in Section 8.3.

Lemma 4. The mapping $M$ under our pub/sub operational semantics guarantees in-order processing of external events (when all events are published from a single source).

Proof. Follows directly from Lemma 3.

Furthermore, our subscription mapping of PAC rules in $\Gamma$ schema also process events with respect to external events (which are published from a single source in both $\Gamma$ and $\Pi$ schemas), which introduces the notion of event-based pseudo-clock (Section 8.5) in order to achieve event-order serialization.

Theorem 3. The mapping $M$ under our pub/sub operational semantics guarantees execution of subscriptions based on the event-order serialization.
Proof. Follows from subscription condition $\psi_{i,PseudoClock}$, which enforces that subscriptions are processed based on the order of external events. The condition $\psi_{i,PseudoClock}$ assures that subscription notification for event $e_i$ is generated only if all notifications for events $e_0 \cdots e_{i-1}$ have already been generated.

**Theorem 4.** The mapping $M$ under our pub/sub operational semantics guarantees the PDG-based serialization of subscriptions.

Proof. The PDG-based serialization of a micro-B-step (a single PAC rule) and a subscription (super PAC rule) is satisfied in pub/sub semantics because the topological sort order of PDG directly encoded in subscription’s condition ($\psi_{PDG}$), which enforces that a subscription’s condition is evaluated only after all attributes in its condition have been stabilized.

With respect to B-step execution, we also prove that toggle-once properties are satisfied under pub/sub semantics.

**Theorem 5.** The mapping $M$ guarantees the toggle-once principle of B-step.

Proof. The toggle-once principle of B-step, which is achieved by PAC rule design, namely, the relation between PAC rule’s prerequisite ($\pi$) and consequent ($\gamma$) such that roughly speaking $\pi \approx \neg \gamma$ and $\pi$ is always evaluated w.r.t. to system snapshot at the outset of B-step after consuming the external event. This relation is also encoded in our subscription definition given by $M_\pi$.

Lastly, to prove the correctness of the overall execution of pub/sub formulation, we introduce the notion of reachable system snapshot: the global state of system after executing a set of external events. Therefore, the correctness of our model after processing set of external events is determined by comparing the information model (captured by the system snapshot) of $\Gamma$ and $\Pi$ schemas; if the two snapshots are identical, then our workflow to pub/sub mapping is correct; false, otherwise.

We distinguish among two level of equivalence semantics when comparing $\Gamma$ and $\Pi$ system snapshot, denoted by $\Sigma^\Gamma$ and $\Sigma^\Pi$, respectively. These equivalence semantics are: weak and strong equivalence. Without loss of generality, we make the following simplification of the internal data structure (schema) of system snapshot for both $\Sigma^\Gamma$ and $\Sigma^\Pi$: we conceptualize $\Sigma^\Gamma$ and $\Sigma^\Pi$ as simply a collection of all data and status attributes given in $\Gamma$ information model. In addition, in $\Sigma^\Pi$, we also employ a versioning semantics for storing this collection, in which the versioning is advanced with respect to external events. Hence, through versioning in $\Sigma^\Pi$, values of data and status attributes are retained separately for each external event while in $\Sigma^\Gamma$, only the latest version of data and status attributes values are maintained.

**Definition 27.** The (partial) system snapshots $\Sigma^\Gamma$ and $\Sigma^\Pi$ are weakly equivalent up to event $e_i$, denoted by $\Sigma^\Gamma \Leftrightarrow_{e_i} \Sigma^\Pi$, iff the value of stable status attributes in both $\Sigma^\Gamma$ and $\Sigma^\Pi$ are equal.

$$\forall \text{attr} \in \Sigma^\Gamma, \tau(\Sigma^\Gamma, \text{attr}) \neq \emptyset \land \tau_{i}(\Sigma^\Pi, \text{attr}, \rho_{e_i}) \neq \emptyset \rightarrow \tau(\Sigma^\Gamma, \text{attr}) = \tau_{i}(\Sigma^\Pi, \text{attr}, \rho_{e_i})$$

**Definition 28.** The (complete) system snapshots $\Sigma^\Gamma$ and $\Sigma^\Pi$ are strongly equivalent up to event $e_i$, denoted by $\Sigma^\Gamma \Leftrightarrow_{e_i} \Sigma^\Pi$, iff all status attributes in both $\Sigma^\Gamma$ and $\Sigma^\Pi$ are stable and are equal.
∀attr ∈ ΣΓ, τ(ΣΓ, attr) ≠ ∅ ∧ τi(ΣΠ, attr, ρei) ≠ ∅ ∧
τ(ΣΓ, attr) = τi(ΣΠ, attr, ρei)

**Theorem 6.** Any reachable system snapshots ΣΓei and ΣΠei for event ei are either weakly or strong equivalent.

**Proof.** The only means to change data and status attributes are through external events and firing of PAC rules, respectively. The data attributes are changed through external events, since both Γ and Π semantics follow event-based serialization, then changes on data attributes must be consistent under both formulations. The status attributes are modified through PAC rules fired within the scope of each external event; again, we showed that under both Γ and Π formulations, PAC rules follow PDG-based serialization. Moreover, we proved the Γ nuance, i.e., the toggle-once principle partly to retain history, can be emulated under our pub/sub formulation. The toggle-once principle is essential in order to avoid infinite firing of PAC rules within a B-step; thus, achieving a finite number of micro-B-step in a B-step. As desired, both Γ and Π results in firing PAC rules and its subscription counterpart in an identical topological sort order; hence, the values of status attributes are also guaranteed to be identical.

Moreover, our pub/sub operational semantics enables concurrent execution of external events in parallel in accordance to the PDG topological sort order. Suppose the topological sort consists a number of levels, where each level is associated to a set of PAC rules, i.e., subscriptions. Thus, as the external event ei propagates through each level, all status attributes associated to visited levels will be stabilized and will be unaffected by execution of subsequent levels of PAC rules. Therefore, granted the attribute versioning is in-place, the new external event ei+1 can process the subscriptions that falls in levels l1-lj−1 while ei is processing level lj. Therefore, inductively, it can clearly be proven that as soon as a one level of topological sort order is processed by one event, the processed level is ready to accept the subsequent event; hence, our pub/sub operational semantics is capable of processing many events in parallel while satisfying event-based and PDG-based serialization requirements.

We are now in position to prove the correctness of Γ formulation in Π abstraction.

**Theorem 7.** The schema Γ under incremental formulation is equivalent to Π, which establishes the correctness of our mapping M : Γ → Π.

**Proof.** The necessary steps in proving the correctness for workflow mapping in terms of system snapshot reachability condition is summarized in as follows: (1) Event-based serialization of B-step (firing all relevant PAC rules) and subscriptions (a super PAC rule) execution (cf. Theorems 1,3); (2) PDG-based serialization of a micro-B-step (firing a single PAC rule) and a subscription (a super PAC rule) execution (cf. Theorems 2,4); (3) The toggle-once principle of B-step is satisfied in our pub/sub semantics (cf. Theorem 5); (4) Weak equivalence of any reachable partial system snapshots ΣΓei and ΣΠei for any event ei (cf. Theorem 6); and (5) Strong equivalence of any reachable complete system snapshots ΣΓei and ΣΠei for any event ei (cf. Theorem 6).

**Theorem 8.** The time complexity of the mapping M : Γ → Π is linear with respect to the number PAC rules and the size of the Γ schema.

**Proof.** We construct a set of application-specific and generic conditions by iterating over each PAC rules exactly once; in addition, we construct a generic condition for every status attribute in Γ.
8.7 Foundation of Distribution

The mapping of data-centric workflow schema to publish/subscribe schema seamlessly provides an abstraction for a robust distribution and parallel execution of each component (e.g., a set of rules and tasks or their mapped subscriptions counterpart) of the workflow over a loosely decoupled and distributed infrastructure, the basis of the pub/sub abstraction. What is unaddressed by pub/sub mapping is how to determine the actual grouping of workflow components in various processing site within the pub/sub infrastructure. Yet another important property of workflow component grouping lies in the ability to easily move subscriptions among pub/sub processing nodes in order to achieve higher-level functionalities such as load balancing, replication, and availability.

One can imagine the two extreme possibilities of grouping: one that every group entails a single subscription (where each subscription is derived as was shown in Sections 8.4, 8.5) or all subscriptions can be placed into a single component. The former approach achieves the highest level of parallelism (in a sense of distributed execution), but suffers substantially from the increased event traffic in order to coordinate and share data among components across various processing sites. While the latter approach becomes a sequential (in a sense of centralized execution\(^6\)), but require zero event traffic for interactions among various components.

Therefore, our goal is to lay a foundation that enables us to study the loosely distribution of the workflow at various level granularity in order to minimize an objective function, e.g., network traffic, while satisfying additional real-world (hard) constraints, e.g., compliance requirements: enforcing parts of a workflow to be completed in particular geographical region, requiring that data must reside in particular region, or following licensing model that charges for shipping data which indirectly forces the execution to be as close as possible to the data. We formulate the workflow distribution in terms of a portable execution unit that can be carried out in a single processing site (P-site). Thus, P-site is a processing site deployed on a given geographical location that is responsible for executing a set of components in a given workflow such that P-site minimizes the objective function and satisfies a given set of hard constraints.

We formally define our problem as follow: given a set of P-sites \( P_i \), and a partial assignment of subset of subscriptions to each of the P-site (hard constraints), then complete assignment such that minimizes the network traffic among the set of P-sites. Furthermore, we require that P-sites are disjoint, namely, a single subscription cannot be assigned to more than one P-site. The solution to our problem is a complete assignment of subscription-to-P-site. Clearly, any arbitrary assignment, starting from the partial assignment is a solution, but not necessarily a one that minimizes the objective function; hence, not an optimal solution. The Figure 8.3 illustrates an instance of our assignment problem, in which we have three P-sites, where each P-site is assigned one subscription \( P_1, P_2 \) and \( P_3 \), respectively, and we have a set of unassigned subscriptions \( S_4 \cdots S_n \). Without loss of generality, we also combine subscriptions with polarity (positive and negative) into a single subscription, i.e., \( S = \{ S_\oplus, S_\ominus \} \).

This assignment problem is formulated as undirected weighted graph \( G = (E,V) \), where each subscription \( S_i \) is represented by a vertex \( v_i \), and there is an edge between two vertices \( v_i \) and \( v_j \) iff the subscription \( S_j \) is interested in events generated by subscription \( S_i \) or vise versa. Also, we have a set of colours, \( C = \{ c_1, \cdots, c_k \} \), where each \( c_i \) corresponds to the P-site \( P_i \). Consequently, the partial colouration (i.e., partial assignment) of the subset of vertices in \( G, V' \subseteq V \), is given by the mapping

---

\(^6\)A central solution that operates on multi-core or multi-processor, in essence, is a distributed execution which would require some form of communication among the various processors (or threads).
function

$$\chi : V' \rightarrow C.$$  \hfill (8.12)

Moreover, we need a cost function to capture the communication cost between two subscriptions (relative to the size of data and protocol messages); thus, each edge \((v_i, v_j)\) of the graph reflects the communication cost flowing between \(v_i\) to \(v_j\). The cost of data flow is given by

$$C_\Delta : E(G) \rightarrow \mathbb{R}^+.$$  \hfill (8.13)

Likewise, the protocol cost is given by

$$C_\pi : E(G) \rightarrow \mathbb{R}^+.$$  \hfill (8.14)

And the total cost function \(C_F\) is given by

$$C_F = (C_\Delta + C_\pi)(E(G)), \hfill (8.15)$$

which is simply computed by summing the data and protocol cost. Therefore, under our formulation, the objective of our assignment problem is to provide a complete colouration of our graph \(G\) while minimizing \(C_F\)

$$\chi : V(G) \rightarrow C, \hfill (8.16)$$

where \(\chi = \chi\) for all \(v \in V'\), such the sum of all edges weights, given by \(C_F\), whose vertices are not of the same colour is minimized. Essentially, the complete graph colouration results in complete assignment, which in turns partitions the graph into \(k\) disjoint sets of vertices such that each set is assigned to a P-site.

Our graph colouration problem can be reduced to the well-known multiway cut (a.k.a. multiterminal cut) problem [61, 62, 73, 208].
Definition 29. Given an undirected weighted graph $G = (E,V)$, a set of terminal $S = \{s_1, \cdots, s_k\} \subseteq V$, a multiway cut is a set of edges whose removal disconnects the terminals from each other. The multiway cut asks for a minimum weight edge set whose removal disconnects the terminals.

The problem of computing minimum weight multiway cut is NP-hard for any fixed size $k$ [61]. However, for $k = 2$, the problem is tractable and can be solved optimally using standard max-flow min-cut algorithm. Furthermore, for $k \geq 3$, there exists a greedy algorithm with $2 - \frac{2}{k}$ approximation guarantee [208]. This greedy algorithm [208] consists of two phases:

1. For each $i = 1 \cdots k$, compute a minimum weight isolating cut $I_i$ for each $s_i$. This cut is computed optimally using max-flow algorithm by construction a new instance of multiway cut problem which consists of only two terminals, namely, $s_i$ and $S - \{s_i\}$.

2. Discard the maximum weight cut $I_j$, and output the union of the rest, denoted by

$$I = \bigcup_{i=1 \cdots k} I_i - I_j.$$

Clearly, $I$ disconnects any pair of terminals, hence, a multiway cut.

In brief, our graph colouration problem can be re-stated as a coloured multiway cut problem, in which multiple vertices can be assigned to the same color. In general, the vertices with the same colours can be merged to form the terminals; hence, there exists a polynomial reduction of coloured multiway cut to classical multiway cut problem.

Theorem 9. The coloured multiway cut problem is NP-hard and can be solved within $2 - \frac{2}{k}$ approximation.

Proof. The proof simply follows from our reduction of the coloured multiway cut problem to the known multiway cut problem. 

In summary, we formalized the general problem of workflow distribution over the pub/sub abstraction as coloured multiway cut problem. We showed that coloured multiway cut is intractable, but there exists a constant factor approximation algorithm for solving it. Arguably, from theoretical perspective, it is interesting to employ a more complex communication cost function in our workflow distribution which collapses all edges leaving from the subscription $S_i$ to all interested subscriptions residing in a different processing site $P$-site because it is sufficient to transmit a message once from $S_i$ to each interested subscriptions in a $P$-site. The collapsing of edges and additionally extending the problem to directed graph, instead of undirected graph, lead to new challenges for future research. However, it must be stressed that these new caveats do not effect the hardness of the problem, namely, the problem remains intractable.

8.8 Summary

In this chapter, we developed a safe distribution and parallel execution of data-centric workflows over the loosely coupled and highly distributed pub/sub systems. To this end, we made the following contributions: we developed a polynomial-time mapping of data-centric workflows into the pub/sub abstraction
to achieve distributed and parallel execution; we proved the correctness of our mapping through equivalence of reachable system snapshots; and proved the hardness of the optimal workflow distribution over the publish/subscribe abstraction and provided a greedy algorithm with a constant factor approximation.
Part IV

Kernel on Reconfigurable Hardware
Chapter 9

FPGA Query Indexing

9.1 Introduction

Efficient event processing is an integral part of growing number of data management technologies such as real-time data analysis [196, 56, 218], algorithmic trading [182], intrusion detection system [56, 77], location-based services [219], targeted advertising [82, 176, 178], and (complex) event processing [220, 12, 14, 74, 18, 67, 130, 36, 175, 176, 82].

A prominent application for event processing is algorithmic trading; a computer-based approach to execute buy and sell orders on financial instruments such as securities. Financial brokers exercise investment strategies (subscriptions) using autonomous high-frequency algorithmic trading fueled by real-time market events. Algorithmic trading is dominating financial markets and now accounts for over 70% of all trading in equities [103]. Therefore, as the computer-based trading race among major brokerage firms continues, it is crucial to optimize execution of buy or sell orders at the microsecond level in response to market events, such as corporate news, recent stock price patterns, and fluctuations in currency exchange rates, because every microsecond translates into opportunities and ultimately profit [103]. For instance, a simple classical arbitrage strategy has an estimated annual profit of over $21 billion according to TABB Group [109]. Moreover, every 1-millisecond reduction in response-time is estimated to generate the staggering amount of over $100 million a year [144]; such requirements greatly increases the burden placed on event processing platform.

Therefore, a scalable event processing platform must efficiently determine all subscriptions that match incoming events at a high rate, potentially up to a million events per second [55]. Similar requirements are reported for event processing in network monitoring services [196].

To achieve throughput at this scale, we propose and evaluate a number of novel FPGA-based event processing designs (Field Programmable Gate Array). An FPGA is an integrated circuit designed to be reconfigurable to support custom-built applications in hardware. Potential application-level parallelism can be directly mapped to purpose-built processing units operating in parallel. Configuration is done through encoding the application in a programming language-style description language and synthesising a configuration uploaded on the FPGA chip [123]. FPGA-based solutions are increasingly being explored for data management tasks [147, 150, 200, 149, 151, 152, 215, 182].

This promising outlook has a few caveats that make the acceleration of any data processing with FPGAs a challenging undertaking. First, current FPGAs (e.g., 800MHz Xilinx Virtex 6) are still much
slower compared to commodity CPUs (e.g. 3.2 GHz Intel Core i7). Second, the accelerated application functionality has to be amenable to parallel processing. Third, the on-/off-chip data rates must keep up with chip processing speeds to realize a speedup by keeping the custom-built processing pipeline busy. Finally, FPGAs restrict the designer’s flexibility and the application’s dynamism\(^1\), both of which are hardly a concern in standard software solutions. However, the true success of FPGAs is rooted in three distinctive features: hardware parallelism, hardware reconfigurability, and substantially higher throughput rates.

Thus, each of our solutions is formulated as a design trade-off between the degree of exploitable parallelism (cf. Fig. 9.1) versus the desired application-level requirements. Requirements considered are: the ease of the development and deployment cycle (flexibility), the ability of updating a large subscription workload in real-time (adaptability), the power of obtaining a remarkable degree of parallelism through horizontal data partitioning on a moderately sized subscription workload (scalability), and, finally, the power of achieving the highest level of throughput by eliminating the use of memory and by specialized encoding of subscriptions on FPGA (performance).

We experiment with four novel system designs that exhibit different degrees of parallelism (cf. Fig. 9.1) and capture different application requirements. In our application context, achievable performance is driven by the degree of parallelism (in which FPGAs dominate) and the chip operating frequency (in which CPUs dominate). Therefore, our solution design space is as follows: a single thread running on single CPU core (PC), a single thread on a single soft-processor (flexibility) \(^2\), up to four custom hardware matching units (MUs) running in parallel in which the limiting factor is off-chip memory bandwidth (adaptability), horizontally partitioning data across \(m\) matching units running in parallel in which the limiting factor is the chip resources and the on-chip memory (scalability), and, lastly, \(n\) (where \(n \geq m\)) matching units running in parallel (with no memory access because the data is also encoded on the chip), in which the limiting factor is the amount of chip resources, particularly, the required amount

\(^1\)e.g., subscription insert and delete operations are not a given.

\(^2\)A soft-processor is a processor encoded like an application running on the FPGA. It supports compiled code written in a higher-level language, like for example C without operating system overhead.
of wires (*performance*).

The ability of an FPGA to be re-configured on-demand into a custom hardware circuit with a high degree of parallelism is key to its advantage over commodity CPUs for data and event processing. Using a powerful multi-core CPU system does not necessarily increase processing rate (Amdahl’s Law) as it increases inter-processor signaling and message passing overhead, often requiring complex concurrency management techniques at the program and OS level. In contrast, FPGAs allow us to get around these limitations due to their intrinsic highly inter-connected architecture and the ability to create custom logic on the fly to perform parallel tasks. In our design, we exploit parallelism, owing to the nature of the matching algorithm (Sec. 9.2), by creating multiple matching units which work in parallel with multi-giga bit throughput rates (Sec. 9.3), and we utilize reconfigurability by seamlessly adapting relevant components as subscriptions evolve (Sec. 9.3).

### 9.2 Event Processing Model

#### Subscription Language & Semantics

The matching algorithm takes as input an event (e.g., market event and user profile) and a set of subscriptions (e.g., investment strategies and targeted advertising constraints) and returns matching subscriptions. The event is modeled as a value assignment to attributes and the subscription is modeled as a Boolean expression (i.e., as conjunction of Boolean predicates). Each Boolean predicate is a triple of either \([\text{attribute}_i, \text{operator}, \text{values}]\) or \([\text{attribute}_i, \text{operator}, \text{attribute}_j]\). Formally, the matching problem is defined as follows:

**Definition 30.** given an event \(\omega\) and a set of subscriptions \(\Omega\), find all subscriptions \(\Omega_i \in \Omega\) satisfied by \(\omega\).

#### Matching Algorithm

The *Propagation* algorithm is a state-of-the-art key-based counting method that operates as follows [74]. First, each subscription is assigned a key (a set of predicates) based on which the typical counting-based inverted index is replaced by a set of multi-attribute hashing schemes. The multi-attribute hashing scheme uniquely assigns subscriptions into a set of disjoint clusters. Second, keys are selected from a candidate pool using a novel cost-based optimization tuned by the workload distribution to minimize the matching cost [74]. The *Propagation* data structure has three main strengths which makes it an ideal candidate for a hardware-based implementation: (1) subscriptions are distributed into a set of disjoint clusters which enables highly parallelizable event matching through many specialized custom hardware matching units (MUs), (2) within each cluster, subscriptions are stored as contiguous blocks of memory which enables fast sequential access and improves memory locality, and (3) the subscriptions are arranged according to their number of predicates which enables prefetching and reduces memory accesses and cache misses [74].

### 9.3 FPGA-based Event Processing

Commodity servers are not quite capable of processing event data at line-rate. The alternative is to acquire and maintain high cost purpose-built event processing applications. In contrast, our design uses
an FPGA to significantly speed up event processing computations involving event matching. FPGAs offer a cost effective event processing solutions, since custom hardware can be altered and scaled to adapt to the prevailing load and throughput demands. Hardware reconfigurability allows FPGAs to house soft-processors—processors composed of programmable logic. A soft-processor has several advantages: it is easier to program on it (e.g., using C as opposed to Verilog which requires specialized knowledge and hardware development tools), it is portable to different FPGAs, it can be customized, and it can be used to communicate with other components and accelerators in the design. In this project, the FPGA resides on a NetFPGA [135] network interface card and communicates through DMA on a PCI interface to a host computer. FPGAs have programmable I/O pins that in our case provide a direct connection to memory banks and to the network interfaces, which in a typical server, are only accessible through a network interface card.

In this section, we describe our four implemented designs each of which is optimized for a particular characteristic such as flexibility in development and deployment process, adaptability in supporting changes for a large workload size, scalability through horizontal data partitioning for moderate workload size, and performance in maximizing throughput for small workload size. Most notably, the distinguishing feature among our proposed designs is the level of parallelism that ranges from running all subscriptions on a single processor (flexibility) to running every subscription on its own custom hardware unit (performance).

### 9.3.1 Tuning for Flexibility

Our first approach is the soft-processor(s)-based solution (cf. Fig. 9.2), which runs on a soft-processor that is implemented on the NetFPGA platform. This solution also runs the same C-based event matching code that is run on the PC-based version (our baseline); thus, this design is the easiest to evolve as message formats and protocols change. In order to maximize throughput of our event processing application, we chose NetThreads [127] as the baseline soft-processor platform for the FPGA. NetThreads has two single-issue, in-order, 5-stage, 4-way multi-threaded processors (cf. Fig. 9.2), shown to deliver more throughput than simpler soft-processors [126]. In a single core, instructions from four hardware threads are issued in a round-robin fashion to hide stalls in the processor pipeline and execute computations even when waiting for memory. Such a soft-processor system is particularly well-suited for event processing: The soft-processors suffer no operating system overhead compared to conventional computers,
they can receive and process packets in parallel with minimal computation cost, and they have access to a high-resolution system clock (much higher than a PC) to manage timeouts and scheduling operations. One benefit of not having an operating system in NetThreads is that packets appear as character buffers in a low latency memory and are available immediately after being fully received by the soft-processor (rather than being copied to a user-space application). Also, editing the source and destination IP addresses only requires changing a few memory locations, rather than having to comply with the operating system’s internal routing mechanisms. Because a simpler soft-processor usually executes one instruction per cycle, it suffers from a raw performance drawback compared to custom logic circuits on FPGAs; a custom circuit can execute many operations in parallel as discussed next.

### 9.3.2 Tuning for Adaptability

In order to utilize both hardware-acceleration while supporting large dynamic subscriptions on both off-chip and on-chip memories, we propose a second scheme (cf. Fig. 9.3). Since FPGAs are normally programmed in a low-level hardware-description language, it would be complex to support a flexible communication protocol. Instead, we instantiate a soft-processor (SP) to implement the packet handling in software. After parsing incoming event data packets, the soft-processor offloads the bulk of the event matching to a dedicated custom hardware matching unit. Unlike the subscription-encoded matching units used in the tuned for performance design, these matching units use low-latency on-chip memories, Block RAMs (BRAMs) available on FPGAs, that can be stitched together to form larger dedicated blocks of memory. The FPGA on the NetFPGA platform [135] has 232 18kbit BRAMs which are partially utilized to cache a subset of subscriptions. Having an on-chip subscription data cache allows event matching to be initiated even before the off-chip subscription data can be accessed. However, our matching algorithm leverages data locality in the storage of dynamic subscriptions, which may be updated during run time, in contiguous array clusters thereby exploiting burst-oriented data access.
feature of the DDR2 (or SDRAM), off-chip memory, while fetching the subscription data clusters. Thus, having an on-chip subscription cache partially masks the performance penalty (latency) of fetching the subscriptions from the off-chip DDR2 memory, which is the main throughput bottleneck of our FPGA-based event processing solution. Therefore, the maximum amount of useful parallelism in this design is limited by the memory bandwidth; in particular, no more than four custom hardware matching units can be sustained simultaneously; any additional matching units will remain idle because only a limited amount of data can be transferred from off-chip memory to on-chip memory in each clock cycle.

In our design tuned for adaptability, we employ a more generalized design that enables the matching units to support a dynamic and a larger subscription workload than can be supported in our designs that tuned either for scalability or performance. Our adaptability design employs the BRAM-based Matching Units (BMUs) which allows a subset of subscriptions to be stored on the on-chip dedicated low latency BRAMs; thus making the design less hardware resource intensive compared to the our pure hardware implementation (tuned for performance design). Furthermore, coalescing dynamic subscription data into an off-chip memory image is achieved using thePropagation algorithm. The resulting subscription data image is downloaded to the off-chip main memory (e.g DDR2 SDRAM) while loading FPGA configuration bitstream. Nevertheless, any hardware performance advantage promised by a FPGA-based design soon dwindles when the data must be accessed from an off-chip memory. We adopt two approaches to reduce the impact of off-chip memory data access latency on the overall system throughput. Firstly, we take advantage of high degree of the data locality inherent inPropagation’s data structure which helps to minimize random access latency. Secondly, to achieve locality subscriptions are grouped into non-overlapping clusters using attribute-value pair as access keys. Therefore, this data structure is optimized for storing large number of subscriptions in off-chip memory. In addition, we incorporate a fast (single cycle latency) but smaller capacity BRAMs for each matching unit to store subset of subscriptions, which helps mask the initial handshaking setup delay associated with off-chip main memory access, i.e., the event matching can begin against these subscriptions as soon as the event arrives; in the meantime the system prepares to setup data access from the off-chip DDR2 main memory.

The stepwise operation of this design is depicted in Fig. 9.3. Upon arrival of an event, the SP transfers (1) the data packets to the input queue of the system. A custom hardware submodule, the DISPATCHER
unit, extracts subscription predicates-value pairs, which are input to hash functions to generate cluster addresses. Cluster addresses are used to look-up the memory locations (2) of the relevant subscription clusters residing both in BMU BRAMs and in off-chip main memory. TheDispatcher then feeds the event (3) and previously computed cluster addresses (4) on the MU Data Bus (common to all BMUs). Next, theMU Driver unit activates all parallel BMUs to initiate matching (5) using on-chip static subscriptions stored in each BMU, while simultaneously queuing up read requests for the off-chip main memory. The transfer (6) of dynamic subscription data between the BMUs is pipelined to avoid stalling the matching units due to data exhaustion. Finally the match results are pushed (7) into the output queue from which the SP transfers the results to the network interface to be sent to the intended host(s).

9.3.3 Tuning for Scalability

The key property of our proposed design tuned for scalability is the horizontal data partitioning that maximizes parallelism (cf. Fig. 9.4). This design offers the ability to adjust the required level of parallelism (which directly translates into matching throughput) by adjusting the degree of data partitioning for a moderate size workload, yet without significantly compromising the feature offered in our adaptability design. It achieves this by fully leveraging the available on-chip (BRAM) memory to partition the globalPropagation’s data structure across BRAM blocks such that each subset of BRAMs is dedicated to each matching unit, in which the matching unit has an exclusive access to a chunk of the globalPropagation’s structure. Unlike our adaptability design in which the degree of parallelism is quite restricted due to the off-chip memory’s access latency, resulting in several data starved or stalled matching units, this design employs matching units (BMUs) (cf. Fig. 9.4) that are each provisioned with a dedicated BRAM memory in order to keep them fully supplied with subscription data. Therefore, the degree of parallelism achieved is simply a function of the number of BMUs that can be supported by the underlying FPGA. Finally, a non-performance critical soft-processor (SP) can be employed to update the on-chip memory tables attached to each BMU in the design; hence, supporting dynamic subscription workload.

The overall stepwise operation of our tuned for scalability design, depicted in Fig. 9.4, is similar to that which occurs in the tuned for adaptability design for steps (1) to (5), with the difference being in the absence of the off-chip main memory used for storing the dynamic subscriptions. The operation and logic of theDispatcher and MU Driver submodule is further simplified as the off-chip memory...
access arbitration and data dissemination to BMUs is eliminated. Every BMU consists of a four-state Finite State Machine, that upon receiving the event data (3) initiates matching by sequentially fetching one subscriptions every clock cycle from the dedicated BRAM memory containing the cluster starting at the address (4) that was dispensed by the Dispatcher unit. Since all BMUs are ran in parallel and in sync with each other, the Dispatcher must dispense the next cluster address only when all BMUs have completed matching all subscriptions in the current cluster. In final phase (6), once all BMUs finish matching all the subscriptions’ clusters corresponding to the predicates present in the incoming event, the final result tallying phase is initiated where matched subscriptions or number of matches found are placed on the match hit vectors and consolidated as a final result value by the Dispatcher unit to be transferred to SP via the output queue.

### 9.3.4 Tuning for Performance

Our final approach (cf. Fig. 9.5) is a purely hardware solution: custom hardware components perform necessary steps involving event parsing and matching of event data against subscriptions. This method provides near line-rate performance, but also involves a higher level of complexity in integrating custom heterogeneous accelerators in which both the performance-critical portion of the event processing algorithm and the encoding of subscriptions are incorporated within the design of the matching unit logic; thereby, completely eliminating all on- and off-chip memory access latencies. Essentially, each subscription is transformed into a self-contained custom hardware unit; this subscription encoding achieves the highest level of parallelism in our design space because all subscriptions are ran in parallel.

The performance design offers the highest rate at which incoming events can be matched against subscriptions, which are encoded in the Subscription Encoded Matching Unit (SEMU) logic on the FPGA. This method avoids the latency of both on and off-chip memory access, but significantly constrains the size of the subscription base that can be supported. A diagram of this design is shown in Fig. 9.5. This setup is massively parallelized and offers event matching at extremely high rates (i.e. one per clock cycle).

The stepwise operation of the our tuned for performance design is depicted in Fig. 9.5. In this design, the soft-processor (SP) only serves to transfer (1) the received event data packets from the network interface input buffer to the input queue of the our system. Custom hardware submodule, the Dispatcher module, parses (2) the incoming events and feeds the current event data to all the matching units while the Mu driver module generates all the necessary control signals to run all SEMUs synchronously. Each unit is able to match all encoded subscriptions against the current event in one clock cycle. However, subsequent clock cycles are spent in tallying the matches and preparing the final responses (e.g. forward address look-up or consolidating system wide match counts) that is eventually pushed (3) into the output queue. The SP then transfers (4) the final result from the output queue to the network interface to be sent to the intended host(s).

### 9.4 Evaluations

This section describes our evaluation setup including the hardware used to implement our FPGA-based event processing system and the measurement infrastructure.
9.4.1 Platform & Evaluation Setting

Evaluation Platform

Our FPGA based solutions are instantiated on the NetFPGA 2.1 [135] platform, operating at 125MHz and have access to four 1GigE Media Access Controllers (MACs) via high-speed hardware FIFO queues (cf. Fig. 9.2) allowing a theoretical 8Gbps of concurrently incoming and outgoing traffic capacity. In addition, a memory controller to access the 64 Mbytes of on-board off-chip DDR2 SDRAM is added. The system is synthesized to meet timing constraints with the Xilinx ISE 10.1.03 tool and targets a Virtex II Pro 50 (speed grade 7ns). Our soft-processor and matching units run at the frequency of the Ethernet MACs (125MHz).

Evaluation Setup

For our experiments, we used HP DL320 G5p servers (Quad Xeon 2.13GHz) equipped with an HP NC326i PCIe dual-port gigabit network card running Linux 2.6.26. As shown in Fig. 9.6, we exercised our event processing solutions from the server executing a modified Tcpreplay 3.4.0 that sends event packet traces at a programmable fixed rate. Packets are timestamped and routed to either the FPGA-based designs or PC-based design. Each FPGA-based design is configured as one of the solutions described in Sec. 9.3 and PC-based is a baseline serving as comparison only. The network propagation delays are similar for all designs. Both FPGA-based or PC-based designs forward market events on the same wire as incoming packets which allows the Event Monitor (EM), cf. Fig. 9.6, to capture both incoming and outgoing packets from these designs. The EM provides a 8ns resolution on timestamps and exclusively serves for the measurements.

Evaluation Workload

We generate a workload of tens of thousands of subscriptions derived from investment strategies such as arbitrage and buy-and-hold. In particular, we vary the workload size from 250 subscriptions to over 100K subscriptions. In addition, we generate market events using the Financial Information eXchange (FIX) Protocol with FAST encoding.

Evaluation Measurements

We characterize the system throughput as the maximum sustainable input packet rate obtained through a bisection search: the smallest fixed packet inter-arrival time where the system drops no packets when

---

3http://www.fixprotocol.org
monitored for five seconds—a duration empirically found long enough to predict the absence of future packet drops at the given input rate. The latency of our solutions is the interval between the time an event packet leaves the Event Monitor output queue to the time the first forwarded version of the market event is received and is added to the output queue of the Event Monitor.

### 9.4.2 Evaluation Results

In this section, we present our evaluation results.

#### Packet Processing

Measuring the baseline packet processing latency of both PC and FPGA-based solutions is essential in order to establish a basis for comparison. When processing packets using the PC solution, we measured an average round-trip time of 49µs with a standard deviation of 17µs. With the NetThreads processor on the FPGA replying, we measure a latency of 5µs with a standard deviation of 44ns. Because of the lack of operating system and more deterministic execution, the FPGA-based solution provides a much better bound on the expected packet processing latency; hence, our FPGA-based solution outperformed the PC-based solution in baseline packet processing by orders of magnitude.

#### Event Processing

Before we begin our detailed comparison of various designs, we study the effect of the number of matching units (MUs) on the matching latency for our scalability design, Table 9.1. As expected, as we increase the number of MUs, moving from 1 MU to 128 MUs, the latency is improved significantly especially for the larger subscriptions workload (with chip resources permitting). This improvement is directly proportional to the degree of parallelism obtained by using a larger number of MUs.

<table>
<thead>
<tr>
<th></th>
<th>1x MU</th>
<th>4x MUs</th>
<th>32x MUs</th>
<th>128x MUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>7.5</td>
<td>5.5</td>
<td>5.0</td>
<td>5.6</td>
</tr>
<tr>
<td>1K</td>
<td>9.3</td>
<td>6.1</td>
<td>4.3</td>
<td>4.3</td>
</tr>
<tr>
<td>10K</td>
<td>64.0</td>
<td>19.0</td>
<td>6.8</td>
<td>5.4</td>
</tr>
<tr>
<td>50K</td>
<td>223.5</td>
<td>59.9</td>
<td>12.3</td>
<td>7.3</td>
</tr>
</tbody>
</table>

Table 9.1: Latency (µs) vs. the # of MUs (Scalability Design)

In Table 9.2, we demonstrate the system latency as the subscription workload size changes from 250 to 100K. In summary, even though our FPGA (125MHz Virtex II) is much slower than the latest FPGA (800MHz Virtex 6) and significantly slower than our CPU (Quad Xeon 2.13GHz), our design tuned for adaptability is 8X faster than the PC-based solution on workload sizes of less than 1K and continued to improve over the PC solution by up to a factor of two on workload of sizes of 100K. Similarly, the design tuned for performance, while currently feasible only for smaller workloads due to lack of resources on the FPGA, is 21.2X faster. Most importantly, our design tuned for scalability takes advantage of both of our adaptability and performance designs by finding the right balance between using the on-chip memory to scale the workload size while using the highly parallel nature of the performance design to scale the event processing power. Thus, the scalability design is 16.2X faster than our adaptability design and is 27.8X faster than the PC design. In addition, a similar trend was also observed for the system throughput experiment as shown in Table 9.3.
Therefore, the *adaptability* design is limited because of slower off-chip memory bandwidth which greatly hinders the degree of parallelism while the *performance* design is limited because encoding the subscriptions in the logic fabric of the chip consumes much more area than storing them in BRAM or DDR2 providing much denser storage. Finally, contrary to general perspective that software solution cannot be utilized in hardware, the success of our *scalability* design (which adapts a software-based solution) suggests that in order to scale our solution to large subscription workloads, certain software data structures for data placement become a viable solution in conjunction with hardware acceleration and parallelism.

### 9.5 Summary

We observe that event processing is at the core of many data management applications such as real-time network analysis and algorithmic trading. Furthermore, to enable the high-frequency and low-latency requirements of these applications, we presented an efficient event processing platform over reconfigurable hardware that exploits the high degrees of hardware parallelism for achieving line-rate processing. In brief, the success of our *fpga-ToPSS* framework is through the use of reconfigurable hardware (i.e., FPGAs) that enables hardware acceleration using custom logic circuits and elimination of OS layer latency through on-board event processing together with hardware parallelism and novel horizontal data partitioning scheme.

In this work, we explored the entire design space with respect to use off- and on-chip memory and purely hardware solution. We present four designs.

1. **Flexible** design consists of a single thread on a soft-processor feeding from slow, large off-chip memory that requires least amount of knowledge of hardware development and enables rapid software development cycle.

2. **Adaptable** design consists of up to four custom hardware matching units (MUs) running in parallel feeding from slow, large off-chip memory that sustains a higher matching rate than soft-processor-based solution but limited to off-chip memory bandwidth.

3. **Scalable** design consists of horizontally partitioning data across \( m \) matching units running in parallel feeding from small, but, fast on-chip memory that the now limited to the size of on-chip memory
(supporting smaller subscription workloads compared to the above approaches) and amount chip resources.

4. *performance* design consists of \( n \) (where \( n \geq m \)) matching units running in parallel (with no memory access because the data is also encoded on the chip) that is limited solely by the amount of chip resources, particularly, the required amount of wires.

This four design not only cover the entire spectrum of high-level designs, but also rule out applicability of any solution which is solely based on soft-processor and off-chip memory, which, although simpler and more flexible, but are unable to harness true processing power of FPGAs. As a result, in our experimental evaluation, our design tuned for *performance* outperformed the design tuned for *flexibility* by a factor of 22X on small size subscription sets while our design tuned for *scalability* outperformed the *flexible* design by a factor of 300X even as the workload size was increased; in fact, this gap further widens as workload size increases due an increased opportunity to process a larger amount of data in parallel.
Chapter 10

FPGA SQL Query Indexing

10.1 Introduction

The need for efficient real-time data analytics is an integral part of a growing number of data management technologies such as intrusion detection [56], algorithmic trading [182], and (complex) event processing [176]. What is common among all these scenarios is a predefined set of continuous queries and an unbounded event stream of incoming data that must be processed against the queries in real-time.

The challenges for today’s real-time data analytics platforms are to meet the ever growing demands in processing large volumes of data at predictably low latencies across many application scenarios. The volume of traffic on the Internet has undergone an immense increase over the last decade which is apparent in deployments of high communication bandwidth links across the globe (e.g., OC192 at 9.92Gbit/s). While according to Gilbert’s law, communication bandwidth is projected to double every 9 to 10 months, conventional computation system architectures are showing signs of saturation in terms of offering the necessary processing power to sustain demands imposed by future Internet bandwidth growths.

The need for more processing bandwidth is the key ingredient in enabling innovation in high-throughput real-time data analytics to process, analyze, and extract relevant information from streams of events. Therefore, as proliferation of data and bandwidth continues, it is becoming essential to expand the research horizon to go beyond the conventional software-based approaches and adopt other key enabling technologies such as reconfigurable hardware in form of Field Programmable Gate Arrays (FPGAs). An FPGA is a cost-effective hardware acceleration solution that has the potential to excel at analytics-based computations due to its inherent parallelism. FPGAs can exploit low-level data and functional parallelism in applications with custom, application-specific circuits that can be re-configured, even after the FPGA has been deployed. In addition, FPGAs can meet the required elasticity in scaling out to meet increasing throughput demands.

We propose an FPGA-based real-time data analytics platform that supports line-rate processing of event streams over a collection of continuous queries. Our contributions are three-folds. (1) We propose high-throughput, custom circuits to implement the relational algebra (i.e., selection, projection, and join) over a window of input events in order to effectively process a single SPJ (Select-Project-Join) query in reconfigurable hardware. The hardware implementation enables a high degree of parallelism and pipelining beyond the reach of software-based implementations. The custom circuits serve as a
library of operators. (2) We introduce a novel multi-query optimization technique inspired by highly parallelizable rule-based system designs by mapping an SPJ-query into a Rete-like operator network [97]. We exploit the overlap among SPJ query plans by constructing a single global query plan to be executed in hardware. (3) We develop software-to-hardware multi-query processing techniques that map a set of SPJ queries into a Rete-like global query plan. Subsequently, the global plan is converted into Hardware Description Language (HDL) code using our “hardware library” of custom building blocks for the various relational algebra operators. These mapping techniques are akin to a compiler that can process a query expressed in our language into a custom circuit that processes event streams.

Moreover, in our design, we exploit parallelism while sustaining the onboard multi-giga bit throughput rates. First, owing to the inherent parallel nature of Rete-like processing, we synthesize custom logic to execute multiple query plans in parallel, while exploiting query plan overlaps. Second, owing to the potential for parallel processing within each relational operator, most notably the expensive join operation, we synthesize custom logic for the operator implementations.

Recent work has shown that FPGAs are a viable solution for building custom accelerated components [138, 89, 147, 150, 200, 149, 151, 215, 152, 182, 199, 142, 214, 216, 192, 33, 201, 58]. In chapter 9.1, we focused on atomic and stateless matching (i.e., select queries). However, in this chapter concentrates on supporting a more general multi-query stream processing (stateful matching) specifically designed to accelerate the execution of SPJ queries. Alternatively, [151] presented an efficient FPGA implementation of a single query (without join) while [199] focused on the data flow for streaming join over a large window size that spans many processing cores. Our approach also differs from [151, 199] as we are primarily concerned with multi-query optimization (with joins computed over a moderate size window) using Rete-like processing networks, supporting a rich relational algebra over event streams, and offering an unprecedented degree of inter- and intra-operator parallelism that is only available through low-level logic design.

10.2 FPGA Event Stream Processing Model

Our event matching data model is captured as attribute-value pairs, which closely resembles a database tuple, but, unlike traditional databases, we do not assume a fixed schemata for the event stream. Similarly, our event stream language also follows traditional database SPJ queries including selection ($\sigma_c$), projection ($\pi$), and join ($\times$). In fact, we adapt PADRES SQL (PSQL) [115], an expressive SQL-based declarative language for registering continuous queries against event streams over an either time-based or count-based sliding window model. Essentially the sliding window is a snapshot of an observed finite portion of the event stream.

Formally, we define stateful matching semantics as follows:

**Definition 31.** Given a stream of events and a collection of continuous SPJ queries, the queries are continuously executed over the event stream.

10.3 Operator-to-Circuit Mappings

The first step to realize query processing on hardware is an efficient mapping of relational operators to custom circuits (custom processors). This mapping forms the basis of our query processing model on
the FPGA. The operator mappings that we discuss are selection, projection, and join. In addition, we explore two circuit designs: the sequential and the parallelized. The sequential solution is tuned for scaling the number of supported queries while the parallel solution (focus of this chapter) is designed for achieving line-rate processing of event streams over a set of queries.

Selection Selection refers to the conditional test over attributes of an event. This test is a unary operation written as $\sigma_c$ where $c$ is a propositional formula over the logical operators $\vee$, $\wedge$ and $\neg$. A combinational circuit is used to implement this propositional formula, referred to as the Selection Circuit ($SC$). One of the key features of $SC$ is that it evaluates the entire propositional formula in one clock cycle. To further accelerate the execution, in our parallelized scheme, we scale out the computation by replicating the $SC$ block $k$ times in order to evaluate up to $k$ events in parallel and in one clock cycle. However, in our resource-aware sequential design, we create a single $SC$ block and the necessary logic to serialize events to sequentially process the selection condition.

Projection Projection refers to the removal of certain attribute–value pairs from an event. Projecting out attributes is implemented as a combinational circuit, Projection Circuit, that uses a mask. A mask has as many bits as the number of attributes in an event such that all of its bits are set to one except those that correspond to the projected attributes. The circuit consists of two-input AND gates that are required to do a bitwise AND operation between an event and the mask.

Join At a high-level, the join operation ($\bowtie_{k}$) consists of two count-based (or time-based) sliding

Figure 10.1: Overview of parallel join processing
windows (left and right window) over two event streams and a join condition $c$ (an arbitrary Boolean expression). The join operation follows the classical window-based-join semantics which is defined as a sequential procedure in two phases: (1) Upon arrival of a new event $e$ at window $w$ (either left or right), $e$ is compared against every other event residing in the opposite window. And for every pair of events that satisfy the condition $c$, the two events are joined and added to the join-result stream. (2) The new event $e$ overwrites the oldest event in $w$. In what follows, we first provide a brief background, then we focus on a novel transformation of this sequential software-oriented procedure into a highly parallelized hardware-oriented implementation by utilizing custom circuits coupled with local on-chip memory banks.

The join computation consists of the join condition $c$ and the Phase 1 of the join semantics which together form the Join Circuit ($JC$); $JC$ is also associated to a left and a right window. The $JC$ block leverages on-chip BRAM memory as the medium for implementing the sliding window. Each block is independent of the others with its own address space and read and write ports. Furthermore, the BRAM port-width can be adjusted to sustain the necessary memory bandwidth, namely, reading and writing of the entire content of either a window ($k$ events) or a BRAM block ($2k \times n$ events), in one cycle when relevant data is stored contiguously. This is referred to as $k$-way read and write, where $k$ is the port-width. To support concurrent read and write, the BRAM is made dual-ported so that reading or writing is carried out on different ports. Finally, to fully utilize the available on-chip BRAM, we must coalesce sliding window buffers from up to $n$ join operations into one of the many available BRAM blocks; thus, packing up to $2k \times n$ events into a BRAM block.

We propose key opportunities for intra- and inter-parallelized execution of window-based-join semantics. For intra-parallelism, first, it is observed that Phase 1 and 2 (of the join semantics) can be done in parallel because the read and write memory accesses are performed on opposite windows; consequently, no race condition occurs and no locking is needed as long as both phases are completed before accepting
additional newly arriving events. Second, Phase 1 can be executed in parallel by comparing the new event \( e \) against every other event in the opposite window in one cycle. This is achieved by replicating the join condition circuitry \( k \) times and enabling \( k \)-way read memory access which in turn yields up to \( k \) joined events. Also, Phase 1 and 2 can be extended to support up to \( k \) simultaneous new events for each window, yielding up to \( k^2 \) joined events in \( k \) cycles.

For inter-parallelism, we can scale up to \( n \) different join operations whose sliding windows are located in a shared BRAM block. Executing \( n \) joins in parallel is possible by replicating the entire join-semantics circuitry \( n \) times and enabling \((2kn)\)-way read memory access. Notably, the machinery for implementing \( n \) joins is tightly coupled to a single BRAM block (i.e., promoting local memory access); thus, no central coordination is necessary among the various BRAM blocks and all can be active simultaneously.

We briefly present data and execution flow of each circuit. The high-level machinery for accepting up to \( k \) simultaneous events for each of the \( n \) join operations (intended for a single BRAM block) is captured in Fig. 10.1, which consists of the following inner-blocks. **BRAM Access Scheduling Block (BASB)** schedules access to sliding window buffer (cf. Fig. 10.2). **Remapping Block (RB)** advances sliding window by evicting the oldest event first (cf. Fig. 10.3). **Join Circuit (JC)** processes parallelized join (cf. Fig. 10.4).

The circuit in Fig. 10.1 captures the overall data and execution flow of \( n \) joins (in what follows we refer to the numbered steps for describing the figure). This circuit accepts as inputs \( k \) new events for either left or right window of every \( JC_i \). The inputs are detected after passing through a combinational
circuit called Check-if-Inputs-Present (CIP) (1). If any event detected for JC\textsubscript{i}, then JC\textsubscript{i} is marked as active and is sent to BASB for execution scheduling (1). All active JC\textsubscript{s} that simultaneously detect inputs are also scheduled to be executed in parallel. If active, JC\textsubscript{i} cannot be executed immediately; then it is temporarily queued (2); otherwise, active JC\textsubscript{s} start receiving the contents of their windows and JC\textsubscript{i}’s \( k \) new incoming events (3). Finally, in two parallel pipelines both active JC\textsubscript{s} are ran in parallel to carry out the join computation (Phase 1 of the join semantics) and for each active JC\textsubscript{i}, RB is invoked to overwrite the oldest events with their corresponding \( k \) new incoming events (Phase 2 of join semantics) (4).

The BASB circuit (cf. Fig. 10.2) simply manages queuing (through a circular array) and scheduling active JC\textsubscript{i} and orchestrating control signals for muxes and demuxes in order to access the BRAM’s content (Steps 1-2).

RB (cf. Fig. 10.3) is responsible for evicting the oldest events when window \( w \) is full. After new events are detected by CIP, (1), events’ ages are stored\(^1\) in on-board registers (2). Next, the indices of the oldest events in \( w \) are extracted (3); these indices become the bases for configuring the crossbar for overwriting the oldest events with the newest ones (Steps 4-5).

The JC\textsubscript{i} block (cf. Fig. 10.4) receives the content of its windows and the same \( k \) new events on every cycle (Steps 1-3). In each cycle, JC\textsubscript{i} takes one of the new events and compares it against all the events in its corresponding window in parallel through replication of JC\textsubscript{i} Join Condition \( k \) times (4). In each

\(^1\)To sort efficiently in hardware, we use Bitonic sort, implemented as combinational circuits that requires a constant number of cycles for small input size, i.e., given \( d \) inputs, Bitonic sort has a comparator stage complexity of \( O(\log^2 d) \) and requires \( O(d \log^2 d) \) comparators.
cycle up to \( k \) joined events are produced \((5)\).

**Time Complexity** The selection and projection operators are implemented as combinational circuits and have complexity \( O(1) \). The join operator complexity is rather involved. In particular, \( RB \) determines eviction of the oldest event and carries out the actual eviction in \( O(1) \), which has to be repeated \( n \) times for each join, resulting in time complexity \( O(n) \). Each \( JC \) evaluates \( k \) input events in \( O(k) \) time (or \( O(1) \) for a single event). The remaining components are executed in constant time such as \( BASB \) for BRAM access coordination; \( CIP \) for detecting input events; and muxes and demuxes for selecting and routing events. Hence, \( n \) join operations (within a BRAM block) can be done in parallel in \( O(\max(k, n)) \).

### 10.4 Multi-query-to-Circuit Mappings

In the previous section, we discussed how to map the building blocks of each query (i.e., the relation algebra) into circuits, thereby paving the way for efficiently processing a single SPJ query on hardware (cf. standard flow in Fig. 10.5). We now shift gears towards processing multiple queries efficiently in hardware, in which the novelty of our proposed approach is to go beyond executing a single optimized query plan on hardware and to support parallel processing of multiple queries on the FPGA (cf. custom flow in Fig. 10.5). An FPGA design is especially powerful in exploiting parallelism because any form of parallel execution can be directly mapped to logic circuits in hardware. In our implementation, this is accomplished by using a Rete-like event processing network to realize a single global query plan (cf. Fig. 10.6) that exploits the overlapping components among given SPJ query plans to further improve the resource utilization and execution of the global query plan on the FPGA.

A multi-query optimized event processing network is comprised of Rete-specific elements, e.g., pattern detect nodes and join nodes, that share functional resemblance with the key relational algebra operators, e.g., \( \sigma \) and \( \bowtie \), respectively, which in turn constitute the elements of a standard SPJ query plan. Hence, a multi-query optimized event processing network, represented as an inverted global SPJ query plan, can ultimately be translated into a hardware design (as per custom flow in Fig. 10.5). The resulting hardware design is modular and in our implementation utilizes the three main rudimentary hardware building blocks (described in the previous section) that realize their relational algebra operator counterparts. The process of mapping a Rete-like graph representing a global SPJ query plan onto a Hardware Description Language (HDL) design (circuit) is captured in our custom flow (cf. Fig. 10.5) which includes a custom Rete-to-HDL compiler. The input to this compiler is a set of SPJ query plan(s) that are firstly used
to build a multi-query optimized Rete network graph using the standard Rete algorithm. Secondly, the resulting Rete graph is decomposed to utilize appropriate HDL models from our custom HDL library for SPJ operators (cf. Section 10.3). Finally, predefined HDL design templates are referenced to build the final circuit that is targeted to execute the given input SPJ query(s) on the FPGA.

To further optimize the global query plan, we utilize a pipelined design that increases chip resource utilization by keeping all processing blocks active at all times. This design delivers a greater throughput than a non-pipelined counterpart at the cost of additional resources. To enable a pipelined design, additional buffering is required in order to avoid dropping events between pipeline stages; furthermore, additional logic circuits are required to orchestrate the flow of events downstream from one operator to the next. A detailed description of the inner workings of the pipelined design is omitted in the interest of the space.

10.5 Platform Architecture and Setup

In this chapter, we describe our FPGA-based event processing platform targeted at processing event streams over a set of continuous queries at line-rate. Our platform provides an “active query window” (cf. Fig. 10.7) to analyze event streams over multiple queries in parallel while streams seamlessly pass through the platform. The filtered output stream is delivered to higher-level applications for further data analytics and monitoring purposes.

The input to our platform is a stream of events, and in our setting, the system throughput, i.e., the maximum sustainable input event rate, can be measured as follows. Due to the direct, unshared Ethernet link, there are no severe transmission reliability issues to consider, except for the filling up of input/output buffers on the board. If no further events can be accepted by the board, packets are dropped and the maximal sustainable processing rate is determined.

Our setup includes one PC that transmits an event stream, over a 1 Gb/s Ethernet interface, to our platform hosted on an Xilinx Virtex 5 LXT ML505 FPGA board (cf. Fig. 10.7). In addition, a USB-JTAG link is employed to program the FPGA board through a second PC loaded with the Xilinx ISE10.1 EDK development tool suite for design synthesis and bit stream generation. Thus, we upload the HDL
design based on user input queries, generated through a SQL-like interface, using our custom Rete-HDL compiler (cf. Fig. 10.5). Finally, along with system level runtime statistics (e.g., query throughput and event buffer utilization), the query output is retrieved from the board via a dedicated secondary link for displaying the results in the “profiler utility” (cf. Fig. 10.7).

10.6 Summary

In this chapter, we proposed hardware acceleration of complex event (i.e., stateful matching) processing. In particular, we develop high-throughput, custom circuits to implement the relational algebra over a window of input events for effective processing of \textit{SPJ} (Select-Project-Join) queries. The hardware implementation enables a high degree of parallelism and pipelining beyond the reach of software-based implementations. We introduced a novel multi-query optimization technique inspired from highly parallelizable rule-based system designs by mapping an \textit{SPJ}-query into a Rete-like operator network while exploiting inter- and intra-parallelism. Therefore, through software-to-hardware multi-query compiler, we first map a set of \textit{SPJ} queries into a Rete-like global query plan. Subsequently, the global plan is converted into Hardware Description Language (HDL) code using our “hardware library” of custom building blocks for the various relational algebra operators.
Chapter 11

Conclusions

11.1 Summary of Results

Query Indexing Kernel

In this thesis, we present BE-Tree, a novel index structure to efficiently index and match boolean expressions defined over a high-dimensional space. BE-Tree’s two-phase space-cutting technique copes with the curse of dimensionality underlying the subscription and event space, which appears in many application domains. Furthermore, BE-Tree’s cost model and self-adjustment policies enables BE-Tree to actively adapt to workload changes. Moreover, we propose scalable and effective predicate evaluation techniques, i.e., lazy and bitmap, and a Bloom filter optimization, which substantially improve BE-Tree’s matching computation by up to 75%. Finally, through an extensive experimental evaluation, we demonstrate the dominance of BE-Tree as a generic index for Boolean expressions that supports a variety of workload configurations and handles predicates with expressive set of operators.

Furthermore, we study the problem of parallel event processing. Particularly, we enhance and parallelize BE-Tree by introducing a novel event stream compression algorithm enabled through a bitmap-based event encoding. Furthermore, we develop an efficient online event stream re-ordering (OSR) approach to exploit the full potential of our adaptive parallel compressed matching algorithm (A-PCM). Our extensive evaluation demonstrates the effectiveness of the proposed A-PCM algorithm that outperforms existing sequential and parallel matching algorithms by a factor of up to 503X.

To address the problem of finding the most relevant matches, we present BE*-Tree, a data structure that employs an effective hierarchical top-k pattern matching algorithm. It is important to note, that BE*-Tree’s top-k model could be applied to BE-Tree; thus, further strengthening the top-k model. BE*-Tree also introduces a novel non-rigid space-cutting technique to index Boolean expressions over a high-dimensional space by developing (1) a bi-directional tree expansion technique that enables indexing only non-empty continuous sub-spaces and (2) a descendant-repelling and overlap-free splitting strategy. Finally, the performance of BE*-Tree is proven through a comprehensive experimental comparison with the index structures for top-k matching of Boolean expressions.

Lastly, to enable the high-frequency and low-latency requirements of our query indexing kernel, we present an efficient event processing platform over reconfigurable hardware that exploits the high degrees of hardware parallelism (including identification of unique intra- and inter-parallelism for window-based-join semantics within multi-query optimization) for achieving line-rate processing. In brief, the success
of our \textit{fpga-ToPSS} framework is through the use of reconfigurable hardware (i.e., FPGAs) that enables hardware acceleration using custom logic circuits and elimination of OS layer latency through on-board event processing together with hardware parallelism and a novel horizontal data partitioning scheme. As a result, our design tuned for \textit{performance} and \textit{scalability} substantially outperforms the PC-based solution.

\section*{Data Indexing Kernel}

The multi-version temporal database market is growing \cite{53}. A temporal database simplifies application development and deployment by pushing the management of temporal logic into database engines. By adopting temporal technologies, the development cost can be reduced by a factor of 10 \cite{53}. This success has led major database vendors (including Oracle \cite{162}, IBM \cite{110}, and TeraData \cite{193}) to provide support for multiversion temporal data.

We tackle a key challenge of multiversion databases: providing good update performance and good query performance in a single system. Transaction throughput and analytical query processing often have conflicting requirements due to the high index maintenance cost for transactions. Our efficient index maintenance using \textit{Indirection} makes indexes more “affordable,” substantially improving the available configuration choices. Our evaluation demonstrates a query cost reduction by a factor of 3 without an increase in update cost. The batching of insertions using our \textit{LiDBlock} technique can save up to 90\% of the insertion time.

\section*{Kernel Extensions}

While most existing XML/XPath matching approaches use standard techniques such as NFAs or DFAs for processing XPath expressions, we altogether diverted and considered a fundamentally new approach for XML/XPath matching. Most importantly, we formalize and introduce a novel XPath encoding based on a general attribute-based Boolean expressions, compatible with our indexing kernel, that supports horizontal and vertical navigation within an XML document and enables structural and value-based filtering. Furthermore, we present a dichotomy of the XPath language that can be mapped into our novel Boolean expression encoding.

To further extend applicability of our indexing kernel, we proposed a safe distribution and parallel execution of data-centric workflows over the loosely coupled and highly distributed pub/sub systems. To this end, we make the following contributions: we develop a polynomial-time mapping of data-centric workflows into the pub/sub abstraction that is aimed at realizing distributed and parallel execution; we prove the correctness of our mapping through equivalence of reachable system snapshots; and prove the hardness of the optimal workflow distribution over the publish/subscribe abstraction and provided a greedy algorithm with a constant factor approximation.

\section*{11.2 Future Work}

In this section, we summarize the future directions of this thesis, considering both angles of our proposed query and data indexing kernel.
A growing challenge in processing analytical type queries in large databases is to cope with the ever increasing volume and velocity (i.e., data arrival rate in form of update, insertion, and deletion queries) of data. Recently, there has been a new paradigm shift for big data processing which is strikingly similar to event processing. In this new paradigm, disks are continuously scanned, and data is fetched in chunks and pushed to only interested queries. Essentially, in this push-based model, a traditional database is transformed into a streaming database which brings two key benefits: elimination of the need for indexing data and relying solely on a fast sequential scan of disks. This scenario is captured in Figure 11.1.

This new model also partially boils down to efficiently identifying which queries are interested in the latest fetched data chunk (or tuple), i.e., indexing queries instead of data. One way to find the interested queries is by extracting a query’s selection conditions that are expressed as Boolean expressions, and in turn indexing these expressions. Examples of such systems are Crescando [206] and DataPath [17]. A promising research direction is to study the role of our indexing kernel in the push-based multi-query optimization especially when data is judiciously pre-compressed on disk and our adaptive compressed parallel matching algorithm is employed.

The indexing kernel can further be enriched in order to support arbitrary Boolean expressions that are not limited to only disjunctive normal form expressions; a requirement demanded in many application scenarios such as push-based multi-query optimization, real-time data analytics, and our proposed data-centric workflow extensions. More importantly, in order to handle the sheer volume of today’s social and enterprise data, the possibility of matching computation distribution over the MapReduce abstraction model is highly attractive.

Another direction in improving the core of our indexing kernel is to explore additional compression technique to further reduce our bitmap encoding size, essentially introducing a second layer of compression. Also our current bitmap-based encoding focuses on discrete domain; therefore, it would be
interesting to apply our discrete domain optimization including adaptive parallel compressed event matching (A-PCM) and online stream re-ordering (OSR) for continues domain (applicable to BE*-Tree) by incorporating our lazy-predicate evaluation model, for example. Most interestingly, our proposed parallel compressed matching algorithm can go beyond stateless matching (i.e., event matching) and enable processing stateful matching (i.e., complex event matching) by aggregation and coalescing a sequence of events (i.e., compressing events). Such aggregation mechanism could model the time- or count-based window-based semantics required in the stateful matching.

A hardware-aware indexing kernel on FPGAs can achieve higher hardware-parallelism through novel (SQL) query encoding and efficient data placement algorithm (through a combination of replication and horizontal data partitioning that conceptually resembles MapReduce style processing) that eliminates chip’s idle areas and resources. More importantly, the problem of adaptive query encoding (changing queries on-the-fly without FPGA reconfiguration) remains open, despite the recent attempts to address this problem [183, 201]. Therefore, we believe, there are many potential opportunities to design novel compressed encoding techniques that exploit a combination of the slow off-chip memory, the fast on-chip registers, and the on-chip logic fabric to achieve line-rate processing while supporting on-the-fly changes of queries. An effective compression technique is the key ingredient to attain this goal because through data compression, a significant memory-bandwidth reduction (the main bottleneck observed on FPGAs) can be obtained while with the abundance of processing-bandwidth available on FPGAs, a better utilization of processing-bandwidth can be achieve if the processing can be carried either directly on compressed data or efficiently decompressing the data as needed.

A prominent direction is to investigate heterogeneous architectures hosting both FPGAs and CPUs. This research directions is further motivated due to emerging FPGA-based co-processors on a single motherboards (e.g., Intel Xeon FSB FPGA Socket Fillers by Nallatech) integrated via high-bandwidth interconnects (64-bit 1066MHz FSB) and with up to 256GB direct system memory access and a 8GB/s peak bandwidth. This is a promising platform for transforming cutting edge for real-time complex data analysis by orchestrating computations across FPGAs and CPUs.

In general, hardware programming is becoming more accessible to the software-oriented minds at large due to a number of research and industry projects that are working on adapting higher-level language (such as Java or C) and runtime environments that dynamically allow (all) portions of an application to be executed on both software and hardware or on developing software-to-hardware compilers. An example of the former projects is IBM’s Liquid Metal [20], that is, based on a Java-compatible language and a runtime that enables execution on both FPGA and GPU [20, 191]. Examples of the latter initiatives for accelerating software algorithms on FPGAs are ImpulseC, C-to-FPGA and OSCI C++-to-FPGA. Building on this success, we are aiming to further improve SQL query indexing on FPGAs, by investigating into a new abstraction-level for expressing queries on FPGAs, namely, FPGA Query Language (FQL). We envision that FQL has an abstraction closer to underlying hardware, thus, less declarative compared to SQL, yet it provides a greater opportunity for developing better automatic query optimizer for FPGA compilation. It is important to note, that our the problem of developing the right query language abstraction is much simpler than the abovementioned initiative (e.g., Liquid Metal [20]) because unlike these approaches that are aiming to solve the general programming optimization problem on hardware, we are narrowing our focus to a particular subproblem of query execution; therefore,
substantially increasing the potential for success.

Other promising directions in FPGA landscape is introducing our indexing kernel into transparent middleware services motivated by observed evolution pattern in today’s enterprise. It is a given that today’s enterprise applications continuously evolve. This is not only due to changing customer requirements, but also due to requirements that stem from application-orthogonal concerns, often readily addressed with quickly available value-added middleware services. Unfortunately, integrating new services into an existing enterprise system, may lead to complex and unpredictable side-effects, such as impact on performance and behavior of the previously running system.

It is exactly this conflict, we aim to address by reconciling the gap between the desired application-orthogonal, value-added services and their unavoidable, costly side-effects. To address this goal, we are planning to develop, a transparent middleware services appliance architecture that is superimposed over an existing enterprise architecture by tapping into the messaging stream between various system components, remaining fully oblivious to the running applications. Our approach exploits reconfigurable logic elements, custom design, combined with low-level parallelism and emerging heterogeneous computing architectures to address the challenge of providing an appliance, built around our indexing kernel, that non-invasively and non-disruptively assists in the provisioning of middleware services.

An example of such a service is enterprise system instrumentation (i.e., application performance management [169]) and monitoring for establishing provenance in order to extract causal relationships and to perform root cause analysis, create audit trails, check for compliance, or detect fraudulent behavior [60, 145]. The service might resort to annotating application messages with lineage and provenance information as they pass through the middleware. This need for maintaining provenance information is a well-known challenge [209, 146, 111, 93]. While supporting these functions – ideally, as value-added services – is critical, the possible side-effects are by and large unavoidable. Side-effects might manifest as impact on system performance, impact on the cohesion and coupling of the system architecture, impact on delivered business functions, and impact on the overall system complexity. This impact must be minimized and should be avoided altogether to truly enable an effective marketplace for middleware services.

**Data Indexing Kernel**

There remain several open problems that we plan to address in future in order improve the data indexing component of our kernel. One primary concern in multi-version databases is space. If every version of a record is copied in full, the space needed to store the database will grow dramatically. There is an obvious potential to compress records, taking advantage of commonality between old and new record versions (as shown using our DeltaBlock technique). Even so, the trade-offs are not straightforward. One does not want to burden transactions or queries on current data with extensive compression/decompression tasks, so a background process that leaves the current record uncompressed but incrementally compresses the historical records may be a good idea. Keeping many versions of a record on a common page may help compression, but would be incompatible with other clustering orders.

Another promising research direction is to introduce SSDs into storage hierarchy of key-value stores such as Google’s BigTable [49], Cassandra [128], HBase [202], and Redis [186]. Although our Indirection technique is directly applicable to key-value stores index maintenance, yet there remains potential opportunities for utilizing SSDs during periodic and expensive routine of merging, purging, and reconstructing the (secondary) indexes in existing key-value store systems.
**Kernel Extensions**

The next key step in our XML/XPath mapping is to increase the expressiveness power of our XML/XPath encoding, thereby generalizing our notion of XPath consistency (Theorem 12) to go beyond single path and enable XPath expressions defined over multiple paths. With regards to data-centric workflow mapping, two research avenues remain open: (1) to implement and to systematically evaluate our proposed mapping over **PADRES** [80, 133], an enterprise-grade pub/sub infrastructure, by extending our existing workflow execution engine [155] and (2) to identify a minimum set of primitives (language constructs) to formulate any data-centric workflows.
Bibliography


