Queries, Data, and Statistics: Pick Two

by

Chaitanya Mishra

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

Copyright © 2010 by Chaitanya Mishra
The query processor of a relational database system executes declarative queries on relational data using query evaluation plans. The cost of the query evaluation plan depends on various statistics defined by the query and data. These statistics include intermediate and base table sizes, and data distributions on columns. In addition to being an important factor in query optimization, such statistics also influence various runtime properties of the query evaluation plan.

This thesis explores the interactions between queries, data, and statistics in the query processor of a relational database system. Specifically, we consider problems where any two of the three — queries, data, and statistics — are provided, with the objective of instantiating the missing element in the triple such that the query, when executed on the data, satisfies the statistics on the associated subexpressions. We present multiple query processing problems that can be abstractly formulated in this manner.

The first contribution of this thesis is a monitoring framework for collecting and estimating statistics during query execution. We apply this framework to the problems of monitoring the progress of query execution, and adaptively reoptimizing query execution plans. Our monitoring and adaptivity framework has a low overhead, while significantly reducing query execution times. This work demonstrates the feasibility and utility of overlaying statistics estimators on query evaluation plans.

Our next contribution is a framework for testing the performance of a query processor by generating targeted test queries and databases. We present techniques for data-aware query generation, and query-aware data generation that satisfy test cases specifying statistical constraints. We formally analyze the hardness of the problems considered, and present systems that support best-effort semantics for targeted query and data generation.

The final contribution of this thesis is a set of techniques for designing queries for business
intelligence applications that specify cardinality constraints on the result. We present an interactive query refinement framework that explicitly incorporates user feedback into query design, refining queries returning too many or few answers.

Each of these contributions is accompanied by a formal analysis of the problem, and a detailed experimental evaluation of an associated system.
Acknowledgements

Nick Koudas is a great supervisor. Thanking him for specifics won’t work; there are too many things. Here is a random sample instead (my research uses sampling too, ha ha): working through twenty poorly written drafts of a paper to get it into shape; scheduling phone meetings at midnight, and waking up at 8 AM on the day of a deadline to help me get a paper out; spending entire afternoons staring at a whiteboard while I struggled to formalize a problem; taking me to some memorable restaurants, and suggesting many more (in places like St. Petersburg, and Dubai!); and getting me to take coffee seriously. Most importantly, he created a great work-hard-but-have-fun kind of environment which made the last five years pretty nice overall.

Cristiana Amza and Hans Arno Jacobsen served on my supervisory committee, asked lots of tough questions and provided useful comments at many checkpoints, and (most importantly) let me pass! Many thanks to Peter Marbach for joining the examination committee for the final defence at short notice. A big thank you to Jignesh Patel as well for serving as my external examiner and providing useful suggestions on the thesis.

This research was generously supported by IBM Canada through an IBM CAS fellowship. Calisto Zuzarte at IBM Toronto served as a useful sounding board for many ideas presented herein. In particular, he identified the applicability of our work on refinement to database testing, which led to our research on the TQGen and TDGen systems.

Maksims Volkovs designed the frontend of ConEx. Ady Ecker pointed us to distinct value estimators, and Amit Chandel formalized a histogram construction for ConEx. Tomasz Nykiel extended ConEx to the map-reduce model. Mokhtar Kandil participated in many discussions where we tried to figure out if ConEx could be used for workload management. Eric Hsu, and Sudipto Guha entertained many questions when I fruitlessly tried to generalize XS to complete query plans. A suggestion from an anonymous referee led to a complete rewriting of the XS paper, and significantly improved its presentation. Volker Markl provided us with the DMV data generator, and along with Peter Haas participated in discussions on extending ConEx and XS. Rick Rashid gave us a “killer application” for Stretch ‘n’ Shrink.

The other denizens of the DBlab were a great group to hang out with. Everyone worked on something different and interesting, and I learnt tons from discussions with them, especially in our friday afternoon meetings. I’ve called on each of them multiple times to drop whatever they were upto and listen to me explain some problem on a whiteboard; they have never refused.

Many many friends worked really hard at ensuring that I always had excuses to avoid research. I’ll name no names but you all know who you are.

My parents let me do whatever I want to, and love me all the same. My sister cheerfully refuses to have serious conversations with me, and I’m eternally grateful for that.

This was fun.
# Contents

1 Introduction 1
  1.1 Queries, Data, and Statistics ........................................... 3
  1.2 Overview ................................................................. 5

2 Background 11
  2.1 The Architecture of a Query Processor .............................. 11
  2.2 Sampling-based Cardinality Estimation .............................. 14

I Missing Statistics 18

3 ConEx: Query Monitoring 19
  3.1 Related Work .......................................................... 21
  3.2 Model and Assumptions ................................................... 23
  3.3 Joins ................................................................. 25
  3.4 Aggregation ............................................................. 40
  3.5 Progress Estimation ..................................................... 44
  3.6 System Design and Implementation .................................... 45
  3.7 Evaluation .............................................................. 46
  3.8 Interface Design ........................................................ 59
  3.9 Conclusions and Future directions ..................................... 62

4 XS: Adaptive Query Processing 64
  4.1 Related Work .......................................................... 65
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Accuracy of aggregation estimators on a 150000 row table</td>
<td>53</td>
</tr>
<tr>
<td>3.2</td>
<td>Memory Overheads</td>
<td>55</td>
</tr>
<tr>
<td>3.3</td>
<td>Overheads in Hash and Merge joins</td>
<td>56</td>
</tr>
<tr>
<td>3.4</td>
<td>Overheads in Index Nested Loops joins</td>
<td>56</td>
</tr>
<tr>
<td>3.5</td>
<td>Overheads of Estimation</td>
<td>57</td>
</tr>
<tr>
<td>6.1</td>
<td>Table Sizes</td>
<td>142</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Problem Statement: Given two, generate the third .......................... 4
1.2 Thesis structure ........................................................................... 9
2.1 Plan for the query Select C.z, Count(*) from A,B,C where A.x=B.x and A.y=C.y group by C.z ................................................................. 13
3.1 Sample query plan ....................................................................... 24
3.2 Join Pipelines ............................................................................ 29
3.3 Multistage Filters Algorithm ....................................................... 37
3.4 System Design ........................................................................... 45
3.5 Binary hash joins ......................................................................... 47
3.6 Comparison of estimators for binary hash joins ......................... 48
3.7 Estimation With large relations ................................................... 49
3.8 Join pipeline with the same attribute .......................................... 50
3.9 Join pipeline with different attributes ....................................... 51
3.10 Approximate summary structures for joins with Z=1 ............... 52
3.11 Aggregation Estimators ............................................................. 54
3.12 Sampling Overheads ................................................................. 58
3.13 Comparison of progress estimators TPCH Q8 1GB Z=2 .......... 59
3.14 Query monitoring in action ....................................................... 62
4.1 Join Pipelines (Outer Input on the Right) and Join Lattice .......... 68
4.2 Join Lattices and Tuple Contribution Dependency Tables ........ 71
4.3 3 Way INL Joins ........................................................................ 81
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>3 Way Hash Joins</td>
<td>83</td>
</tr>
<tr>
<td>4.5</td>
<td>Multiple Queries</td>
<td>84</td>
</tr>
<tr>
<td>4.6</td>
<td>4 Way Joins</td>
<td>85</td>
</tr>
<tr>
<td>4.7</td>
<td>Multiway Joins (TPCH)</td>
<td>86</td>
</tr>
<tr>
<td>5.1</td>
<td>Maximal relaxations and contractions for numeric predicates</td>
<td>101</td>
</tr>
<tr>
<td>5.2</td>
<td>Query Refinement: Phase 2. Every point on the curve in ( O^0 ) dominates ( k ) points</td>
<td>103</td>
</tr>
<tr>
<td>5.3</td>
<td>Navigation paths for 2 hierarchies (S,C) and (G,A)</td>
<td>108</td>
</tr>
<tr>
<td>5.4</td>
<td>Single Query Varying Target</td>
<td>115</td>
</tr>
<tr>
<td>5.5</td>
<td>Multiple Query Varying Targets</td>
<td>116</td>
</tr>
<tr>
<td>5.6</td>
<td>Varying # Distinct Values for relaxation</td>
<td>116</td>
</tr>
<tr>
<td>5.7</td>
<td>Varying dimensions and skew</td>
<td>117</td>
</tr>
<tr>
<td>5.8</td>
<td>Execution Times</td>
<td>118</td>
</tr>
<tr>
<td>6.1</td>
<td>Sample Test Case</td>
<td>121</td>
</tr>
<tr>
<td>6.2</td>
<td>TQGen Algorithm</td>
<td>132</td>
</tr>
<tr>
<td>6.3</td>
<td>System Architecture</td>
<td>141</td>
</tr>
<tr>
<td>6.4</td>
<td>Single Table Accuracy Experiments</td>
<td>142</td>
</tr>
<tr>
<td>6.5</td>
<td>MultiQuery Accuracy Experiments</td>
<td>143</td>
</tr>
<tr>
<td>6.6</td>
<td>2 Table Accuracy Experiments</td>
<td>143</td>
</tr>
<tr>
<td>6.7</td>
<td>Varying parameters</td>
<td>145</td>
</tr>
<tr>
<td>6.8</td>
<td>Execution Times</td>
<td>146</td>
</tr>
<tr>
<td>7.1</td>
<td>Testing Plan</td>
<td>152</td>
</tr>
<tr>
<td>7.2</td>
<td>Buckets and Cells for 2 test queries</td>
<td>154</td>
</tr>
<tr>
<td>7.3</td>
<td>Top-down Data Generation</td>
<td>165</td>
</tr>
<tr>
<td>7.4</td>
<td>System Design</td>
<td>170</td>
</tr>
<tr>
<td>7.5</td>
<td>Single Table: Varying Data Size</td>
<td>172</td>
</tr>
<tr>
<td>7.6</td>
<td>Single Table: Varying Number of Queries</td>
<td>173</td>
</tr>
<tr>
<td>7.7</td>
<td>Multiway Joins</td>
<td>175</td>
</tr>
<tr>
<td>7.8</td>
<td>Accuracy: Selections</td>
<td>176</td>
</tr>
<tr>
<td>7.9</td>
<td>Accuracy: Joins</td>
<td>177</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Relational database systems are an exercise in abstraction. Data is exposed as relations, which are manipulated through a data manipulation language, and queried using a declarative query language like SQL. As far as the user is concerned, the physical data layout, query processing architecture, data size, and even the backend system architecture (a single computer or a large cluster) is immaterial; for all practical purposes, the database system acts as a black box. User programs remain unchanged whether relations are stored as rows or columns, whether the query processor supports hash joins or nested-loops joins, and whether the relation is stored on a laptop or on a large cluster.

The simple, yet powerful idea of abstracting data as relations relies on two assumptions to be successful.

- The interface is powerful enough for user needs. For database systems to be successful, it is essential that most user needs can be easily and intuitively captured by the declarative querying interface.

- The system can provide acceptable performance. Relational database systems require the user to give up a certain degree of control on data layout and query execution. For this to be acceptable, the performance of the system should be comparable to hand-coded user programs, while providing the additional benefit of a clean relational interface.

Both these assumptions have to a large extent been validated by the success of relational database systems, which now constitute a multi-billion dollar market. Database vendors such as
Oracle, IBM, and Microsoft, are some of the largest and most successful software firms around. A relational database system is one of the core components of almost any modern application stack.

Recent trends however are challenging the validity of the assumptions stated above. These include:

- **Analytical Queries**: Traditionally, database systems focused on transactional data processing. Queries were simple and short running, while the major challenge was ensuring ACID semantics for transactions. Increasingly however, businesses have started maintaining data warehouses on which they process analytical queries. These queries are often complex, involving multiple tables. They also process large amounts of data, and can consequently run for hours. Moreover, such queries are often ad-hoc and exploratory, being crafted by analysts to extract interesting information.

  A consequence of analytical queries is that the user often has little or no idea as to whether the query will run for minutes or days, and whether the query will return millions of tuples, or an empty answer. The simple interface implemented by traditional database systems provides users with no feedback about the progress of the executing query, and about the expected properties of the query result. Moreover, complex ad-hoc queries are prone to optimization errors, since they involve multiple tables \[76\] and may utilize attributes on which statistics are not maintained by the system. To summarize, when a user submits a complex analytical query to a database system, the user has no idea as to the

  - progress of execution of the query.
  - properties of the result, such as its size.
  - optimality (or lack thereof) of the query evaluation plan.

- **Complex Data**: Modern databases are often very complex, with many large tables and attributes with skewed data. As a case in point, compare the decision support benchmarks from the Transaction Processing Performance Council \[4\]. The current benchmark (TPC-H) \[3\] was established in 1999, with 8 tables containing attributes with uniform data, and 22 benchmark queries. The draft version of the new proposed decision support benchmark (TPC-DS) \[2\] has 24 tables, attributes with skewed data, and 99 benchmark queries drawn from a mix of query types. The minimum acceptable database size for TPC-H benchmarking was 1 GB; for TPC-DS it is 100 GB.

  Complex data, when coupled with analytical queries, makes it difficult for database systems to achieve optimal performance. This is a consequence of the large number of options for maintaining statistics, indexes, and materialized views on the data. Moreover, skewed and correlated data defies the traditional assumptions of uniformity and independence which
statistics estimators for query optimization rely on. Consequently, query plans are often sub-optimal. However, existing query processors provide no support for detecting and switching inefficient plans during query execution.

- **Complex Systems:** Database systems were designed according to the principle that complexity should be hidden from database users and application developers. A consequence of this design decision is that the systems themselves are complex. A modern data management system contains, among other components: a disk manager, a buffer manager, a scheduler, a query processor with many operators, a query optimizer, and a transaction manager. This complexity leads to problems for database administrators (DBAs), and system developers/testers.

For example, achieving good performance on a large database installation requires the DBA to configure several parameters such as the buffer cache size, temp space, multiprogramming level etc. In addition to these global parameters, the DBA must also tune indices, materialized views, and statistics. Database systems provide minimal information to help DBAs make these decisions; database tuning is almost a black art. In an ideal scenario however, database systems should require minimal or no administration, adapting autonomically to the data and workload.

The complex internals of a database system make it challenging for system developers and testers to exhaustively test its performance. As noted previously, even the new version of the decision support benchmark (TPC-DS) has only 99 benchmark queries. Running this benchmark tests only a fraction of the use-cases of the system. This suggests the need for automatic test query and data generation frameworks that satisfy ad-hoc test case specifications.

We claim that modern database applications are severely challenging the assumptions on which database systems are designed: the sufficiency of the declarative interface, and the guarantee of achieving good performance. We claim that the weakening of these assumptions affects the usability of database systems. In this thesis, we take steps towards bridging the gap between the vision of database systems that simply work, and the reality of today’s database engines. We introduce techniques to augment the traditional database interface, and explore techniques to improve the performance of database query processing. We study these problems by exploring the interdependence between queries, data, and statistics in a relational query processor.

### 1.1 Queries, Data, and Statistics

The query processor of a relational database systems executes declarative SQL queries on data represented as relations. Queries are executed using query evaluation plans, which are typically trees of physical operators. The evaluation plan is usually generated by a cost-based optimizer [114],
which takes into account estimates of various statistics associated with the query and data. Such statistics of interest include base, intermediate, and final relation sizes; the number of distinct values on a group of columns; and more generally, the distribution of data in a column or group of columns. These statistics impact the execution time of the evaluation plan. For example, a query evaluation plan with small intermediate relations will, ceteris paribus, execute faster than a plan with larger intermediate relations. Similarly, an aggregation operator will typically execute faster on a grouping column with fewer distinct values, even if the underlying relation size remains the same. Estimating and exploiting the statistics associated with the query and data is at the heart of modern query processing and optimization.

Given a query $Q$, a database $D$, and a statistic of interest $S$, obtaining the value of $S$ is straightforward: we simply generate the subexpression $Q_S$ of $Q$ corresponding to $S$, and execute $Q_S$ on $D$. Estimating $S$ without executing $Q_S$ is significantly more challenging. This statistics estimation problem, is however extremely well studied, with (to give a few examples) solutions using histograms [75], wavelets [100], and sampling [62, 88, 59]. These solutions are primarily in the context of query optimization though; statistics are collected offline, and are utilized by the query optimizer to select query evaluation plans.

We assert that many problems associated with query processing can be abstractly formulated as problems that explore the interdependence between queries, data, and statistics in a database system. Examples include:

- **Monitoring:** Any model of progress of query execution requires a model of work, which is a
cumulative function of the work performed by each operator in the plan. The work performed by an operator is a function of its input and output cardinality. Monitoring and predicting the progress of query execution is therefore an online statistics collection and estimation problem.

- **Query Design:** The SQL interface to a database supports a boolean retrieval model, in which constraints are specified on individual tuples. This presupposes that one has knowledge of the precise set of tuples desired. On a large unseen database, users need help in designing queries to extract useful information from the database. Statistics on the data can serve as a guide towards query design and enable exploration of the database.

- **Adaptivity:** Adapting query execution requires identifying whether the currently executing query plan is suboptimal, and identifying a better evaluation plan. As with traditional query optimization, a major component of this problem is statistics collection and estimation. Unlike query optimization though, the statistics collection and estimation needs to be performed in an online fashion, during query execution.

- **Testing:** Testing the performance of an operator in a query plan requires testing its behaviour across various input and output sizes, and data distributions. This requires generation of test cases (queries and data) that satisfy a desired set of statistical properties.

This thesis tackles various problems in query processing that can be abstracted as the problem of estimating, or guaranteeing certain statistical properties. More precisely, this thesis deals with multiple data management problems that can be described by the following meta-problem:

**Problem Statement:** Given two of the following three: query $Q$, data $D$, and statistic $S$; and a template for the unspecified element (either $Q$, or $D$, or $S$), is it feasible to instantiate the unknown element such that query $Q$, when executed on the data $D$ satisfies the statistic $S$?

The class of problems considered is illustrated in Figure 1.1. Given two elements of the triple, we seek to instantiate the missing third element. The specific variations of the problems considered depends on the associated real world problem. In the next section, we present an outline of the thesis, and describe the problems considered in more detail.

### 1.2 Overview

This is a thesis in 3 parts. Each part corresponds to a different unknown in the meta-problem described previously. In Part I, we consider problems where the statistics are unavailable, while the query and data are given. Likewise, Part II considers problems where the query is missing, and in Part III, the data is missing. The technical content in these three parts is bookended by a chapter providing background material (Chapter 2) and a concluding chapter (Chapter 8).
In Part I of the thesis, we assume that the query, and data are given, and estimate statistics associated with the query and data. We consider this problem in an online setting, with the objective of monitoring and estimating statistical properties associated with a query evaluation plan at execution time.

Monitoring Query Execution

Chapter 3 presents the Continuous Explain (ConEx) framework for monitoring and estimating the progress of execution of SQL queries. This has applications as a user interface tool, as well as serving as a generic monitoring substrate for other components of the database system. We demonstrate how the progress monitoring problem can be modeled as an online cardinality estimation problem. ConEx demonstrates how sampling based estimation can be overlaid on a regular query evaluation to accurately estimate intermediate relation sizes during query evaluation. ConEx exploits properties of operators in the query evaluation plan, pushing-down estimation to sorting and hashing phases of operators, obtaining accurate estimates early in the execution of a plan.

In terms of our abstract formulation, the problem considered in Chapter 3 is:

<table>
<thead>
<tr>
<th>Given</th>
<th>Query:</th>
<th>As submitted by the user.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data:</td>
<td>Base tables on which the query is executed, and intermediate data observed at operators during query execution.</td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td>Statistics:</td>
<td>Cardinality of intermediate relations in the query evaluation plan.</td>
</tr>
</tbody>
</table>

Some results from Chapter 3 were published as a short paper at ICDE 2007 [101], and the ConEx system was demonstrated at SIGMOD 2007 [107]. A more detailed description of ConEx was published in TODS 2009 [105].

Adaptive Query Processing

ConEx provides a general framework for monitoring and estimating the progress of queries. In Chapter 4, we present Execution Simulation (XS), an adaptive query processing technique that builds upon ConEx. XS enables detection and correction for optimization errors at execution time through lightweight reordering of join pipelines. XS extends the estimators of ConEx to simultaneously estimate the cardinality along multiple alternative execution plans without making any independence assumptions. This process, which we refer to as simulation, enables XS to identify the optimal reordering of a join execution pipeline during query evaluation.

At its core, XS utilizes an estimation scheme that solves the following problem:
1.2. Overview

<table>
<thead>
<tr>
<th>Given</th>
<th>Query:</th>
<th>As submitted by the user.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data:</td>
<td>Base tables on which the query is executed, and intermediate data observed at operators during execution of the provided query evaluation plan.</td>
</tr>
<tr>
<td>Find</td>
<td>Statistics:</td>
<td>Cardinality of intermediate relations in a class of alternative execution plans considered by the XS reoptimizer.</td>
</tr>
</tbody>
</table>

Note how XS requires cardinality estimates for intermediate relations that may not be observed in the current query evaluation plan. In this aspect, it extends the estimation framework of ConEx, which focuses on the existing query plan. Together, Chapters 3 and 4 provide a unified framework for incorporating sampling based cardinality estimation into a relational query processor. We demonstrate the utility of these changes through applications in query progress monitoring (ConEx) and adaptive query processing (XS).

Results from Chapter 4 on XS were published at ICDE 2009 [104].

1.2.2 Part II: Missing Queries

In Part II of this thesis, we assume that the data and certain statistics are given. Our objective is to instantiate a query template such that the resulting query satisfies the specified statistics when executed on the given data. Specifically, we consider as statistics of interest, the final output and intermediate relation sizes of the query.

Query Design

In Chapter 5 we explore the problem of refining a query to satisfy a cardinality constraint on its output size. This is motivated by Business Intelligence applications, where analysts pose ad-hoc queries on databases, often with a result size in mind. We present the Stretch ‘n’ Shrink (SnS) system for interactive user-aided refinement of SQL queries to a given result cardinality through transformations of the selection predicates. SnS handles both equality and range predicates, and provides a unified solution for query relaxation (increasing the cardinality of the query), and query contraction (decreasing the cardinality) problems.

The abstract problem considered in Chapter 5 can be expressed as

<table>
<thead>
<tr>
<th>Given</th>
<th>Data:</th>
<th>Tables in the database</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statistics:</td>
<td>Desired cardinality of the query result.</td>
</tr>
<tr>
<td>Find</td>
<td>Query:</td>
<td>A query that satisfies the desired result cardinality, generated through predicate transformations on the original user query.</td>
</tr>
</tbody>
</table>

A preliminary version of the SnS system from Chapter 5 was demonstrated at SIGMOD 2008 [102]. Results from this chapter were also published at EDBT 2009 [103].
**Test Query Generation**

*SnS* attempts to generate queries satisfying a single constraint: the output size. Database system developers and testers, on the other hand, often require queries for performance testing that satisfy *multiple* properties at intermediate relations. In Chapter 6, we formally analyze the hardness of this problem, and extend *SnS* to design the *TQGen* system for *targeted query generation*. *TQGen* accepts as input a test case specification, and an underlying database, and generates a query that best satisfies the test case when executed on the database.

In abstract terms, *TQGen* solves the following problem:

<table>
<thead>
<tr>
<th>Given</th>
<th>Data:</th>
<th>Tables in the database</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statistics:</td>
<td>Desired cardinality at multiple nodes in the query evaluation plan.</td>
</tr>
</tbody>
</table>

| Find | Query: | A query that produces the desired number of tuples at the associated intermediate nodes in the query evaluation plan. |

Together, Chapters 5 and 6 provide a means of generating queries that satisfy (one or more) cardinality constraints. While we prove the hardness of this problem, we present best-effort solutions that provide acceptable error in practice.

Results from Chapter 6 were published at SIGMOD 2008 [106].

### 1.2.3 Part III: Missing Data

**Test Database Generation**

In the final part of the thesis, we assume that the query and statistics are given, while the data is to be generated. Specifically, we study the problem of generating a *single* test database satisfying *multiple* test cases defined by queries associated with cardinality and distributional constraints on intermediate subexpressions. In Chapter 7, we formally analyze the complexity of this problem, and describe a practical tool *TDGen* as a solution. *TDGen* incorporates procedures for efficiently generating test data satisfying the cardinality constraints, while providing best-effort results for distributional constraints. We note that this problem is different from the testing problem considered in Chapter 6, which considered only cardinality constraints, and did not handle distributional constraints.

In terms of our framework, Chapter 7 looks at the following problem:
1.2. Overview

Given

<table>
<thead>
<tr>
<th>Queries:</th>
<th>Multiple queries conforming to the same template.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistics:</td>
<td>For each query, and each intermediate node in a common query evaluation plan: the associated node cardinality, and data distributions on attributes relevant to processing at the node.</td>
</tr>
</tbody>
</table>

Find

| Data:       | A single test database such that each query satisfies its associated cardinality and distributional constraints. |

1.2.4 How to read this thesis

Each chapter of the core of this thesis (Chapters 3 to 7) is self-contained, and can be read in isolation. However, the adaptive query processing techniques of chapter 4 utilize the estimation framework developed in chapter 3, and are best understood in that context. Likewise, the targeted query generation framework of chapter 6 incorporates elements from the interactive query refinement technique of chapter 5.

As an alternative structure, the research areas investigated in this thesis, and the associated chapters are illustrated in Figure 1.2. We describe these areas in more detail next:

- **User Interfaces:** Chapter 3 describes a framework for continuously monitoring and predicting query execution. This is fundamentally different from the previous user interface provided by database systems, in which the user is only shown results when they are produced (potentially after hours). Similarly, chapter 5 introduces an interactive interface for query refinement,
that takes into account user preferences. The framework enables interactive formulation of the query using the result cardinality as a guiding target.

- **Autonomic Databases:** The monitoring framework of chapter 3 provides a generic monitoring substrate that can potentially be utilized by various autonomic database components such as schedulers and resource allocators. The adaptive query processing framework of chapter 4 builds upon this monitoring technique, providing a means for query processors to adaptively correct for optimizer mistakes, and improve query execution times.

- **Database Testing:** Chapters 6 and 7 provide two alternative viewpoints towards how to test a database system for performance. The query generation technique of chapter 6 provides a way to generate queries given an underlying test database, and a set of desired properties. The database generation technique of chapter 7 inverts the problem, and looks at how one can generate test databases given test queries and desired properties.

- **Performance:** Chapter 4 provides a way to improve query processor performance by correcting for the errors introduced by the query plan; our technique has been measured to reduce query execution times by up to a factor of 8. Providing a different perspective on performance, the testing techniques of chapters 6 and 7 enable one to test the performance of a query processor under controlled conditions.
In this chapter, we provide background material useful for understanding the research presented in this thesis. Chapter-specific background, and related work is provided in each chapter. Here, we focus on background material that is common to multiple chapters.

In Section 2.1 we describe a model of a typical relational query processor. Specifically, we present the abstractions and interfaces on which relational query processing architectures are built. This material is useful for understanding the motivation of, and techniques presented in chapters 3, 4, 6 and 7.

Following this, in Section 2.2 we provide a brief overview of sampling based cardinality estimation techniques. We utilize these techniques as a basic building block for the material in parts I and II of this thesis, i.e chapters 3, 4, 5 and 6.

2.1 The Architecture of a Query Processor

In this section, we describe the typical architecture of the query processing module of a database system, running on a uniprocessor. Parallel query processing engines have similar architectures; however, in this thesis we focus on uniprocessors.

The query processor of a relational database system processes query evaluation plans over tables of data to generate the output of a query. At an abstract level, these plans are trees of operators. Each operator in the plan performs some simple transformation on its input data to produce its output. Data flows from the leaves of the tree up to the root, which produces the output of the query.
Each operator of an evaluation plan takes as input one or more (typically, at most two) sets/sequences of input tuples, and produces one set/sequence of output tuples. Most operators follow set semantics; however some impose sequence semantics on their output (e.g. a sort or index-scan operator), or input (merge-join). The plan exploration module of the query optimizer ensures that the set/sequence semantics of an operator are respected. The pioneering query optimization paper [114] captured this through the notion of interesting orders.

Given a query evaluation plan representing a tree of operators, the query processor schedules the execution of the individual operators and ensures correct data transfer between them. Each operator implements three functions: `open`, `close` and `getnext`. The `open` function encapsulates the initialization phase of an operator, with any required data structures being initialized. Similarly the `close` function encapsulates the final cleanup phase of an operator. Most importantly however, the `getnext` function encapsulates an iterator interface implemented by each operator. Each invocation of the `getnext` call results in the operator returning a unit (typically, a tuple) of output data.

A consequence of the iterator interface supported by each operator is that query execution proceeds in a demand driven fashion. Query execution begins when a `getnext` call is made on the root of the operator tree. The root operator, in order to satisfy the `getnext` call, performs possibly multiple `getnext` invocations on its child operators, until it has sufficient data to produce its first output tuple. The child operators in turn invoke their children, and this process is repeated all the way down to the leaf nodes, where the tables are accessed.

Such iterator based interfaces are the standard abstraction on which most modern database query processors are built [49]. They were first formalized in the context of extensible query processing architectures such as Volcano [50], Starburst [55, 89] and Genesis [13]. The iterator abstraction enables seamless introduction of new operators in the query processor. As long as the new operator implements the interface correctly, it can easily be integrated into query evaluation plans.

The iterator interface does not mandate how an operator produces its outputs, or how it schedules its children. Most operators implement a lazy scheduling strategy: they perform only the minimal number of invocations of their children to produce an output tuple. For instance, a filter operator with the predicate \( x < 50 \) will keep invoking its child until either it gets a tuple which has \( x \) less than 50, or the child runs out of tuples. In either case, control is transferred to its parent operator until the next invocation of a `getnext` call. The advantage of lazy scheduling is that it minimizes the amount of state to be maintained by the operator. Some operators however, must process their entire input to produce even a single output tuple e.g. a sort operator. In this case, producing the first output tuple is expensive; further output tuples can be easily produced by reading a temporary file. We refer to operators that must process their entire input before producing any output as blocking operators. Other operators are non-blocking.

Blocking operators partition a query evaluation plan into pipelines of non-blocking operators.
2.1. The Architecture of a Query Processor

A pipeline is the maximal set of operators that execute together. The execution of a pipeline is driven by one or more driver nodes, which feed tuples into the pipeline. In join pipelines, driver nodes are the outermost input of the pipeline.

**Example 1.** Consider the query plan illustrated in Figure 2.1 for the query `Select C.z, Count(*) from A, B, C where A.x = B.x and A.y = C.y group by C.z`. The query is executed in three pipelines. Pipeline 1 scans the table A, and sorts the tuples on A.x. Pipeline 2 performs a Merge join on A and B on attribute x, and pipelines the results to a Nested-Loops join operator, which probes an index on C.y to perform the join with relation C. The results are fed to a sort operator on C.z. Finally, Pipeline 3 processes the sorted output of the sort operator, and groups the data on the column C.z. At any point in the execution, only one of the pipelines is active.

The notion of splitting query evaluation plans into pipelines is at the core of the ConEx framework for query monitoring (Chapter 3), and the XS technique for adaptive join reordering (Chapter 4). Our work on testing in chapters 6 and 7 is also motivated by the need to test query processors using query evaluation plans that satisfy runtime statistical properties on intermediate results.
2.2 Sampling-based Cardinality Estimation

In this section, we provide a brief overview of sampling-based cardinality estimation techniques, as applicable to this thesis. Note that sampling is not the only means used towards obtaining cardinality estimates; however, it is the one used in this thesis since we many of the problems considered herein require online cardinality estimates. For an overview of histograms, the other major technique in database cardinality estimation, please consult the overview paper by Ioannidis [75].

In the following, we describe random sampling based estimators for various primitive relational operations.

2.2.1 Selections

Consider a relation $R$ with size $|R|$. Our objective is to estimate the selectivity of a predicate $\phi$ i.e $\mu_{\phi} = \frac{|\sigma_{\phi}(R)|}{|R|}$. Suppose we have access to a stream of tuples $r_1, r_2, \ldots, r_n, \ldots$ from $R$. This stream is generated by uniform random sampling from $R$ without replacement. Each tuple $r_i$ either passes or fails the predicate $\phi$. Let

$$I_{\phi}(r_i) = \begin{cases} 1 & \text{if } \phi(r_i) = \text{true} \\ 0 & \text{otherwise} \end{cases}$$

$I_{\phi}$ is an indicator function for the set of tuples that pass the predicate $\phi$. Given this, $\hat{\mu}_{\phi}$ defined below is an unbiased estimator of $\mu_{\phi}$

$$\hat{\mu}_{\phi} = \frac{1}{n} \sum_{i=1}^{n} I_{\phi}(r_i)$$

Given this estimator, we would like to obtain confidence bounds on its accuracy. To do so, we require the variance of $I_{\phi}(x)$. Let $Var_{\phi}$ denote the variance i.e $Var_{\phi} = \frac{1}{|R|} \sum_{x \in R} (I_{\phi}(x) - \mu)^2$. Given the random sample, we can estimate $Var(\phi)$ using the estimator $\hat{Var}_{\phi}$ defined below:

$$\hat{Var}_{\phi} = \frac{1}{n} \sum_{i=1}^{n} (I_{\phi}(r_i) - \hat{\mu}_{\phi})^2$$

$$= \hat{\mu}_{\phi}^2 + \frac{1}{n} \sum_{i=1}^{n} (I_{\phi}(r_i))^2 - 2\hat{\mu}_{\phi} \frac{1}{n} \sum_{i=1}^{n} (I_{\phi}(r_i))$$

$$= \hat{\mu}_{\phi}^2 + \hat{\mu}_{\phi} \frac{1}{n} \sum_{i=1}^{n} (1 - 2I_{\phi}(r_i))$$

The estimator $\hat{Var}_{\phi}$ can easily be updated incrementally using the formula above. Since we are sampling uniformly without replacement, we treat $I_{\phi}(r_i)$ as an i.i.d random variable. For large enough $n$, we can apply central limit theorems that assert that $\hat{\mu}_{\phi}$ approximates a normal distribution with mean $\mu_{\phi}$ and variance $Var_{\phi}$. Given this, we can compute confidence intervals with probability $\alpha$ as:

$$P(\hat{\mu}_{\phi} - z_{\alpha} \sqrt{\hat{Var}_{\phi}} \leq \mu_{\phi} \leq \hat{\mu}_{\phi} + z_{\alpha} \sqrt{\hat{Var}_{\phi}}) \approx \alpha$$

(2.1)
2.2. Sampling-based Cardinality Estimation

where \( z_\alpha \) can be obtained from standardized normal tables.

The estimator above defines an online cardinality estimation scheme for selection queries. The procedure continues to process random samples from \( R \) until a desired confidence bound is achieved. Both \( \hat{\mu}_\phi \) and \( \hat{\text{Var}}_\phi \) can be updated in an online fashion as data is processed. This is an essential feature for such online estimation schemes.

Sampling based estimators were first introduced in database systems through the work of Hou et al. [69, 70]. They motivated the use of such estimators in the context of real-time processing of \texttt{count(*)} aggregates on relational algebra expressions. They provided estimators for expressions containing selections, intersections, and joins. In the following section, we describe how the estimator defined above carries over to the case of joins.

2.2.2 Joins

Joins differ from selections since they involve estimating the cardinality of an expression defined over \textit{multiple} relations. Moreover, generating a random sample of the result of a join is known to be a hard problem [31]. To estimate the size of the output of a join, we need to consider various sampling options. For example, whether to sample from each relation or to sample from just one and join the sample with the other relations in full. Haas et al. [59] provide an overview of the various techniques for estimating the size of join using sampling and tradeoffs in using each. In the following, we describe a specific form of sampling which we utilize in multiple chapters of the thesis.

Consider a join of two relations \( R \bowtie_{R.t=S.t} S \). As in the previous section, suppose we have access to a stream of tuples \( r_1, r_2, \ldots, r_n, \ldots \) from \( R \) generated by uniform random sampling without replacement. In the case of selections, each tuple either passed or failed the predicate \( \phi \). In this setting, suppose each tuple \( r_i \) joins with \( N_S(r_i) \) tuples of \( S \) i.e \( N_S = |r_i \bowtie S| \). Our objective is to estimate \( \mu_J = \frac{|R \bowtie S|}{|R|} \). The estimator \( \hat{\mu}_J \) defined below is an unbiased estimator of \( \mu_J \)

\[
\hat{\mu}_J = \frac{1}{n} \sum_{i=1}^{n} N_S(r_i)
\]

Note how this estimator is almost identical to the corresponding estimator for selection operations \( \hat{\mu}_\phi \), with the only difference being that the indicator function \( I_\phi \) is replaced by a join cardinality \( N_S \). Given this estimator, confidence bounds similar to those presented in Equation 2.1 can be derived using the same procedure. As before, we can compute the variance as:

\[
\hat{\text{Var}}_J = \frac{1}{n} \sum_{i=1}^{n} (N_S(r_i) - \hat{\mu}_J)^2
\]

As before, half-width confidence intervals with probability \( \alpha \) can be computed as \( z_\alpha \sqrt{\hat{\text{Var}}_J} \).
2.2.3 Distinct Value Estimators

In the following section, we describe estimation procedures for estimating the number of distinct values in a column $R.x$ of a relation $R$ using only a random sample of $R$. Estimating the number of distinct values in a relation using a random sample is a well studied problem in the database and statistics literature [60, 25, 62, 17].

One of the simplest estimators for the number of distinct values is the GEE technique (Guaranteed Error Estimator) [25]. GEE has guaranteed accuracy as it can provide a lower and upper bound guarantee for its estimate. It operates based on the principle that data can be split into low-frequency and high-frequency groups and that any reasonably sized sample should contain all the high frequency groups. It treats the singleton values in the sample as a sample of the low frequency values and scales them up to minimize the error. If $f_i$ is the number of groups that occur exactly $i$ times in the sample of size $n$ from $R$, the GEE estimate for the number of distinct values in a relation of size $|R|$ is

$$D_n = \sqrt{\frac{|R|}{n} f_1 + \sum_{j=2}^{n} f_j}$$

The estimate follows by taking the geometric mean of the lower and upper bound estimate for the number of singleton groups ($f_1$ and $\frac{|R|}{n} f_1$ respectively).

An alternative estimation procedure is a maximum likelihood estimator $MLE$ of the number of distinct values in a relation [17]. Assume that the column $R.x$ contains $g$ distinct values $v_1, \ldots, v_g$ (i.e $g$ groups). Moreover assume that the fraction of each group $v_i$ in the stream is $p_i$ i.e $p_i = |\sigma_{R.x=v_i}R|/|R|$. After observing $n$ tuples of $R$ our expectation for the number of new (currently unseen) groups that will be identified if we read $\ell = |R| - n$ more values is:

$$\sum_{i=1}^{g} (1 - p_i)^n (1 - (1 - p_i)^\ell) = \sum_{i=1}^{g} (1 - p_i)^n - \sum_{i=1}^{g} (1 - p_i)^{n+\ell}$$

Using this formula as an estimator is not possible, however, since the number of groups $g$ and the values of $p_i$’s are unknown in $R$. However, if the $p_i$’s have low variance, the maximum likelihood estimates (MLE) for both $g$ and the $p_i$’s will be good estimators, since we observe tuples from $R$ in random order. The number of groups observed is $\hat{g} = \sum_{i=1}^{M} f_i$ where $M$ is the maximum frequency count for any group. Similarly, our MLE estimate $\hat{p}$ for a group that has been observed $j$ times in $n$ values from $R$ is $\frac{j}{n}$. As a result, the expectation for the number of groups to be observed if we read $\ell$ more tuples from $R$ would be:

$$D_n = \hat{g} + \sum_{i=1}^{M} f_i (1 - \frac{i}{n})^n - \sum_{i=1}^{M} f_i (1 - \frac{i}{n})^{|R|}$$

This estimate is monotonic and converges to the correct value, as $n$ increases. Moreover, through an asymptotic analysis, it can be shown that such MLE estimators, have bounded variance [17]. This estimator rarely overestimates the number of groups, although it is prone to underestimation.
The GEE and MLE procedures described above are only two of the many distinct value estimation procedures proposed in database and statistics literature. For instance, Ozsoyuglu et al. [111] propose the following estimator:

$$D_n = \hat{g} + \frac{f_1^2}{2f_2}$$

which estimates a lower bound on the number of distinct values.

Haas et al. [62, 60] provide an extensive overview of various distinct value estimation procedures. In addition, they introduce a hybrid estimator that combines estimation procedures due to Shlosser [115] and a Jackknife estimator [22].

We do not provide a comprehensive review of all possible distinct value estimation procedures. The estimators described above are selected primarily due to the ease with which they can be incrementally maintained with increasing random sample size. This is an essential property for any online estimation procedure to be used during query processing.
Part I

Missing Statistics
Query monitoring refers to the problem of observing and predicting various parameters related to the execution of a query in a database system. Typical parameters of interest are the cardinality [33, 29] or size [92, 93, 94] of the output of operators in the query. The primary challenge in query monitoring is to design observation and estimation techniques that are accurate while incurring a low overhead.

Query monitoring systems provide information that can be of great help to both database administrators and end users. Given the complexity of queries in decision support applications, it is common for queries to take hours or days to terminate. In addition it is also possible for very similar queries to have vastly different execution times, due to the data distribution, and choice of plans selected by the optimizer. These factors make it challenging for even experienced users to predict the execution time of arbitrary queries. A query monitoring system can greatly aid one’s understanding of the progress of a query towards completion and allow one to plan accordingly (e.g., terminate the query and/or change the query parameters). Similarly, from the point of view of administrators, unsatisfactory progress of queries may point to bad plans, poor tuning or inadequate access paths. When a system is overloaded, an administrator can use dynamic query monitoring information to decide which queries are consuming resources and which queries are close to completion. This can be used to make decisions about which queries should be terminated to bring the system back to a normal state. Over a period of time, fine-grained information can be obtained by a query monitoring system and logged for analysis of resource bottlenecks, and further tuning of the system.

In addition to providing feedback to human observers, query monitoring also serves as a basic
building block for enabling adaptive and autonomic techniques in database systems. For example, query reoptimization systems [82, 98, 11] have a monitoring component that observes the cardinality of different operators and estimates the cost of the currently executing query plan. Similarly systems for statistics management such as LEO [118] and SITS [19] require query monitoring tools to collect statistics at intermediate nodes in the query evaluation plan, and optimize future queries. Similarly, cardinality estimates obtained by monitoring query execution can be used by an adaptive resource allocation strategy to allocate resources to the currently executing query.

In this chapter, we develop a lightweight online estimation framework for query monitoring. We introduce several techniques for estimating the cardinality of various operators in the query plan. By monitoring query execution at select points in the plan and collecting statistics from the tuples seen, our framework can predict the output size of operators. Our estimates become increasingly more accurate, converging to the true values as the query executes and we are, under general distributional assumptions, able to analytically quantify their accuracy. We take special care to make our statistics collection and estimations lightweight. We experimentally demonstrate that our framework, when implemented in a real database management system, imposes minimal overhead on query execution time, while enabling highly accurate estimation of query progress.

We also design an interface that presents information in an accessible manner. Typically, the visual interface advocated for monitoring query progress is a progress bar based interface [92]. We argue that such an interface is too simplistic to provide informative feedback about the progress of query execution. Instead, we extend the traditional EXPLAIN interface found in database systems to provide continuously updated information on the status of query execution. We term this system ConEx (Continuous Explain).

To summarize, in this chapter we make the following contributions:

- We present techniques to estimate progressively the size of various types of relational (binary and multiway) joins. We describe the types of guarantees one can obtain and we show how these estimators can be efficiently realized and implemented inside a relational engine. We demonstrate how to tune the computation of our estimations in a way that imposes minimal overhead to query performance.

- We present techniques to progressively estimate the number of groups during GROUP-BY and related operations. Our estimates provably converge to the correct values. We identify the limitations of current state of the art estimators in this area and we present a new hybrid estimator which combines features of existing estimators. We take special care to make our techniques minimally intrusive to query performance in this case as well.

- We implement our techniques in a real relational engine, reporting on our experience and the technical challenges. Additionally, we present the results of an experimental evaluation, using benchmark data, reporting on the accuracy and overheads of our techniques.
3.1. Related Work

We design an interface to present information collected during query monitoring in an accessible way. Our interface exploits the traditional EXPLAIN interface in database systems and displays information at multiple levels of detail.

The rest of the chapter is organized as follows. In Section 3.1 we review previous research related to this problem. Section 3.2 presents some background material necessary for the remainder of the chapter. In Section 3.3, we present strategies for online cardinality estimation for different kinds of join operators. Section 3.4, presents our techniques for aggregation operators. We combine these techniques to present our overall framework for progress estimation in Section 3.5, with a description of the system design in Section 3.6. In Section 3.7 we experimentally evaluate the performance, overheads and accuracy of our framework when implemented inside a real database management system. Section 3.8 presents the design of our interface for query monitoring. Finally, Section 3.9 concludes the chapter.

3.1 Related Work

Tools for monitoring the progress of tasks have been studied in various contexts (e.g. HCI [108], program compilation [91] and file downloads) but there exists limited work on this topic in a data management context.

Recent work on query progress indicators [92, 93, 33, 29, 101] introduced several novel ideas. Chaudhuri et al. [33] introduced the idea of decomposing a query plan into a number of segments (pipelines) delimited by blocking operators. Focusing inside each segment, they use the total number of \texttt{getnext} calls made over all operators during the run of the query as an indicator of query progress. They use the progress at the \textit{driver node} (the node that feeds tuples into the pipeline) as an indicator of the progress made in the pipeline, and use relational confidence intervals to refine optimizer estimates for cardinalities of pipelines that are yet to begin. Subsequent work [29] extended this approach proposing estimators that are worst case optimal, and showed that it is not possible to provide non trivial deterministic guarantees on progress estimation in the general case.

An independent but related approach was presented by Luo et al. [92]. This approach also makes use of segments for query partitioning. The model of work adopted is bytes processed at the input and output of segments. Since the bytes processed at these points is directly related to the cardinality of the operator which demarcate the segment, this measure of progress is analogous to the driver node estimator and the \texttt{getnext} model of progress [33]. Subsequently their framework [93] was extended to a broader class of SQL queries, and introduced refinements that capture the work done within operators in a more fine-grained manner. Although we phrase our presentation in terms of the \texttt{getnext} model, our techniques are equally applicable to this model as well.

In subsequent work, Luo et al. [94] also introduced the notion of multi-query progress indicators. The primary challenge in this work is to devise a model of work that accurately models the
interaction of multiple queries simultaneously executing in the system. In this chapter, we focus on accurately monitoring and predicting the behaviour of a single query. However, we note that our query cardinality estimation techniques can equally apply to any reasonable model of progress in a multi-query setting.

One of the primary goals of our query monitoring system is to provide online feedback about the status of queries submitted to the system. Online aggregation techniques [58] also aim to address the problem of lack of feedback during query execution by taking a different approach. In their work, aggregate values are progressively approximated in an online fashion as the query executes and a continuously refined approximate answer to the query is presented. Accuracy guarantees are provided at any point of the execution and the confidence to the final estimate improves as execution progresses. We provide similar probabilistic guarantees on the quality of our estimation procedures.

A major difference between ConEx and online aggregation lies in the requirement for specialized operators in online aggregation, such as Ripple joins [57]. Despite recent advances [81, 80] ripple joins do not easily scale to data sizes larger than memory. Our present work attempts to perform cardinality estimation without introducing specialized variations of operators in query plans. This is feasible because unlike online aggregation, our objective is not to approximate the final query result, but to estimate statistical properties of the outputs of operators in the plan.

Recent work on adaptive query processing and optimization [82, 98, 11] makes use of statistics collection at various points in the query plan for reoptimization purposes. Specifically, the POP system [98] counts the number of tuples output by an operator and compare this number with a precomputed range of values (a check point) that guarantees the optimality of the remaining plan. Similarly, the Re-Opt technique [82] also collects the number of tuples processed by an operator and use these updated statistics to call a query optimizer to reoptimize the query. While both these techniques have monitoring components, they do not utilize any estimation techniques other than the ones used by the query optimizer. In contrast, the Rio system [11] uses random sampling in order to re-optimize proactively during query execution by getting cardinality estimates in pipelines during query execution. However, the estimation framework is quite limited as it addresses only the case of primary-foreign key hash joins. Our query monitoring framework can be used to aid adaptive query processing techniques by providing online estimates for different operators in the query execution plan. Indeed, in Chapter 4, we describe the XS framework for adaptive query processing, that builds upon ConEx.

Our work builds upon previous research for join cardinality estimation using random sampling [61, 59]. However, this research primarily concentrated on cardinality estimation for the purposes of query optimization i.e before the query starts execution. Online aggregation [58] considers the problem of sampling based cardinality estimation (in the form of COUNT queries) in an online setting, but is restrictive in its choice of query evaluation plans. In this work, we remove the restriction on query evaluation plans, and identify positions where cardinality estimation can
be conducted during query processing. This restriction of not modifying query evaluation plans implies that we cannot read in relations out of order i.e before the plan starts reading the relation. As a result, sophisticated techniques such as bifocal sampling [45] cannot be applied in our setting since they require access to both join relations for estimation purposes.

Similarly, for aggregation operators, a large body of work exists on estimation of the distinct number of values of an attribute (e.g., see [60, 25] and references therein). The estimator (GEE) proposed in [25] has nice analytical properties with optimality guarantees. However, it may lead to severe under/over estimates depending on the distribution. A heuristic estimator (AE) proposed in [25] combines features of various estimators, but isn’t suitable to implement inside the execution engine since it requires invoking numerical methods to solve equations obtaining the right parameters for the estimator. As a consequence, AE is not incrementally maintainable, and hence is not designed for an online setting.

The problem of statistics estimation in data streams is related to our problem. Both the problems feature an online estimation challenge; the difference is that our problem is defined on fixed size base tables, while stream estimation problems are typically defined on sliding windows over unbounded streams. Reviewing the entire literature on estimation over streams would take up too much space; however, problems of interest include: histogram construction [54], join cardinality estimation [7], heavy hitters [42, 96, 84, 39] and quantile identification [97, 52]. Our work builds upon streaming literature, especially on the heavy hitters problem, in Section 3.3.6.

### 3.2 Model and Assumptions

In this section, we describe the measure of progress adopted, and provide definitions necessary for the remainder of the chapter.

In accordance to previous work [92, 93, 29, 33] we assume query plans consisting of a tree of physical operators. The set of operators we consider are **scan**, σ, π, ⊲ ⊳ **nested-loop**, ⊲ ⊳ **index nested-loop**, ⊲ ⊳ **hash**, ⊲ ⊳ **merge**, **outer-join** , **sort** and γ (**group by**). As stated in Section 2.1 query plans can be partitioned into multiple **pipelines**. Pipelines are defined as maximal subtrees of concurrently executing operators. For example, in Figure 3.1, the shaded region consists of one pipeline (a merge join and the index scans feeding it), and the non shaded region consists of another (a hash join, and its probe input). We refer to Section 2.1 for a discussion on the architecture of a query processor, and how plans are structured.

We adopt the model of progress introduced in [33] which is essentially based on the number of getnext calls invoked in a query tree. Throughout this text, the terms getnext model, and gnm are used interchangeably to refer to this model. The notion of query progress introduced by [92] is similar to gnm and our discussion applies to progress estimators using this model as well.

Suppose that the execution of a query Q involves m operators. For each operator O_i, we denote
the total number of \texttt{getnext} calls to the operator over the entire execution of the query as \( N_i \). At any given point during query execution, let \( K_i \), be the number of \texttt{getnext} calls made thus far. The \textit{gnm} measure of progress is then:

\[
\frac{\sum_{i=1}^{m} K_i}{\sum_{i=1}^{m} N_i}
\]

We denote the current number of \texttt{getnext} calls made as \( C(Q) = \sum_{i=1}^{m} K_i \), and the total number of \texttt{getnext} calls over the lifetime of the query as \( T(Q) = \sum_{i=1}^{m} N_i \).

Two observations are in order here. First, \( C(Q) \) can be computed easily by just observing and counting the number of tuples produced by all operators in the plan. Secondly, estimating \( T(Q) \) is equivalent to the problem of cardinality estimation. This problem, despite the large volume of research devoted to it, remains a challenge for general and fairly complex query plans (e.g., see [77]). Thus, under the \textit{gnm} model, the progress estimation reduces to the problem of estimating and refining \( T(Q) \) as the query executes. Previous work on progress estimation [92, 93, 29, 33] split this problem into the problem of refining the estimate of cardinalities of operators in the \textit{currently executing} pipeline, and the problem of refining the estimate of cardinalities for operators in \textit{future} pipelines. Our focus in this work is on faster and more accurate estimations for the \textit{currently executing} pipeline.

We note here that our choice of the \textit{gnm} measure of progress is primarily driven by the simplicity of the measure and the fact that it can be generalized to any measure of progress that relies on the cardinality estimate of queries. For example the estimate of progress used introduced by Luo et al. [92] can be expressed as:

\[
\frac{\sum_{i=1}^{m} w_i K_i}{\sum_{i=1}^{m} w_i N_i}
\]

where \( w_i \) is the average size of tuples produced by operator \( i \). More complicated models of query progress that handle multiple queries [94] can also be rewritten in terms of cardinality estimates of operators. In these models, the weights associated with each operator may vary depending on the number of queries currently executing in the system. In general the problem of defining an accurate model of work is closely related to the problem of modeling the cost of different operators for the purposes of query optimization. However, these cost functions typically take cardinality esti-
mates as input and therefore, progress estimation is linked to the problem of cardinality estimation irrespective of the specific model of work used.

Our framework does not require, but can make use of base table statistics. Such statistics are commonly histograms of the attribute value distribution of single base table attributes. We also assume knowledge of the size of base tables, which is usually available in the system catalogs. In order to provide confidence guarantees on the estimates, we require that table scans on base relations have access to a random sample of the relation (either precomputed or obtained on demand). Our techniques make use of such samples of a specific size to derive cardinality estimates. Once the random samples are processed, base tables are scanned (in the order determined by the plan) and the query is processed accordingly. Similar requirements have previously been proposed in the context of online aggregation [58] and reoptimization [11].

We note that our techniques do not require that entire relations be read in a random order from disk. However, to obtain confidence bounds on our estimates, we require that random samples of certain inputs to the pipeline are available. These requirements vary depending on the structure of the pipeline and the operators in it. Query pipelines read in their various inputs from base tables or temporary intermediate tables produced during query processing. For base tables, our estimation procedures read in precomputed random samples stored on disk and utilize them for cardinality estimation. For intermediate relations produced at blocking points, random samples are computed at runtime by utilizing reservoir sampling techniques [121]. These samples are utilized for cardinality estimation for pipelines higher up in the query evaluation plan. Previous research on progress estimation [92, 93, 33, 29] does not take advantage of random samples. As a consequence, as shown by Chaudhuri et al. [29], it is not possible for them to provide constant factor bounds for progress estimation. By interleaving random sample processing with query evaluation, we are able to obtain probabilistic confidence guarantees on the quality of our estimates. In the absence of random samples, our techniques still provide useful estimates. However, the associated confidence bounds are no longer valid. For the purposes of the discussion that follows, we utilize uniform random samples in our estimation framework.

We now present our techniques for online cardinality estimation for joins, and then proceed on to aggregation operators in Section 3.4.

### 3.3 Joins

Recall the estimation procedure for joins described in Section 2.2.2. We join two relations $R \Join S$, and have access to a stream of tuples $r_1, r_2, \ldots, r_n, \ldots$ from $R$ generated by uniform random sampling without replacement. Each tuple $r_i$ joins with $N_S(r_i)$ tuples of $S$ i.e $N_S = |r_i \Join S|$. Our objective is to estimate $\mu_J = \frac{|R \Join S|}{|R|}$. The estimator $\hat{\mu}_J$ defined below is an unbiased estimator of
$\mu_J$

$$\hat{\mu}_J = \frac{1}{n} \sum_{i \in 1...n} N_S(r_i)$$

Confidence bounds associated with this estimator are provided in Section 2.2.2. This basic formula can be easily adjusted for the case of join conditions involving disjunctions and conjunctions of multiple attributes. On the case of joins with conjunctions, we can treat the multiple join columns together as one single column, with tuples matching if the values along all the join attributes match. In the context of join cardinality estimation, the attribute value $i$ essentially denotes a set of tuples that match with each other. Therefore, it does not matter if we have multiple join conditions. Likewise, the case of disjunctions can be treated by rewriting the expression appropriately as a sum of conjunctive conditions. For instance:

$$|R \bowtie_{R.a=S.a \lor R.b=S.b} S| = |R \bowtie_{R.a=S.a} S| + |R \bowtie_{R.b=S.b} S| - |R \bowtie_{R.a=S.a \land R.b=S.b} S|$$

More general conditions can handled using the principle of inclusion and exclusion as illustrated by Hou et al. [69]

We note here that the cardinality estimation estimation scheme defined here is not necessarily the only scheme that can be utilized in our estimation framework. Our current scheme defines guarantees for join cardinality estimation using random sampling. This is a well studied problem in sampling literature [61, 59, 56] and confidence intervals have been derived using extensions of Hoeffding’s inequality, and central limit theorems. These techniques and their associated analysis can be applied in our problem context as well. On similar lines, signature based schemes for join cardinality estimation [7] maintain a small synopsis of each relation and use it for cardinality estimation. These too can be utilized as estimators in our framework. Our primary contribution in this work is not the introduction of new cardinality estimation schemes, but rather the identification of positions in query plans where such cardinality estimation schemes can be utilized effectively. In the following sections, we describe how these techniques can apply with different join operators.

### 3.3.1 Binary Hash Joins

Assume that the join operation defined previously is a hash join operator. Let $S$ be the build input, and $R$ be the probe input. Suppose that while partitioning $S$, we count the number of times each value in the join attribute is seen. This operation can be interleaved with the actual partitioning to keep overheads low (we provide implementation details in Section 3.7). In other words, we build a histogram that maintains a count $N_v^S$ for each value $v$ in the join attribute $S$. Let $D_n$ be the estimate of the of the join output after $n$ tuples are read i.e $D_n = \hat{\mu}_J \times |R|$

We can express $D_{n+1}$ in terms of $D_n$. If the $(n+1)$th tuple of $R$ the join attribute $v$, we have

$$D_{n+1} = \frac{D_n \cdot n + N_v^S |R|}{n + 1}$$
Notice that this formula does not require histogram construction on the probe input. In addition, it also avoids the expensive operation of multiplying corresponding buckets. For each tuple of \( R \), the histogram built on the join attribute of \( S \) is probed to obtain \( N_v^S \) and update the estimator as detailed above. The computational overhead of this operation is negligible; the only overhead is storage for the histogram on \( R \) (see Section 3.7.2).

An important advantage of using this estimator is that it converges to the exact cardinality of the join by the end of the first pass on the probe input. The hybrid hash join and the Grace hash join algorithms partition the probe input on the first pass into buckets, and leave most (or all in the case of Grace hash join) of the join processing to the second pass. In contrast the driver node estimator (\( dne \)) of [33] and the estimator of [92] do not guarantee convergence until the entire join has been processed. Also, since our technique performs estimation in the first pass of the join, it does not face the problems caused by clustering of the probe input. Recall, that due to hash partitioning all tuples with the same value are clustered into the same partition. The actual join is done by reading data partitionwise, and so the \( dne \) [33] estimate of the output may fluctuate significantly if certain partitions contain tuples that join with many tuples of the build relation while others contain tuples that join with very few tuples. Similar problems can be observed with the estimator of [92]. Our estimator sidesteps this problem of reordering of probe input since it conducts estimation in the hash partitioning pass itself.

We note here that the histograms constructed on the join column of the build input are stored in memory, and are therefore separate from the hashed partitions of the relation, which may be spilled back to disk. This is particularly common for large relations for which algorithms such as the hybrid-hash and Grace joins have been developed. We note that even if the relations are much larger than memory, the space required to store the histogram on just the join column is significantly less. This is because our estimator requires only the distribution of the join column, and does not store entire tuples in memory. Given memory constraints, we utilize approximate histograms [75] to store the distribution of the join column of the build relation. We describe our approach for this problem in more detail in Section 3.3.6 This decoupling of the data (which is partitioned and written back to disk) and the histogram (which is in memory) ensures that these estimation techniques can be used for both hybrid hash and Grace join algorithms.

### 3.3.2 Binary Sort-Merge Joins

The estimator described above can be used for the case of sort-merge joins as well. Consider a sort-merge join between two base relations \( R \) and \( S \) (\( R.a = S.a \)). Suppose \( S \) is sorted first. In the sort operator, every tuple of \( R \) is seen at least once before any output is produced. Thus, it is possible to build a histogram on the join attribute of \( S \) which maintains the number of times each value is seen in the join attribute \( S.a \). Note that this can be done even if the sorting is done within the sort-merge join and not in some separate sort operator. Once this histogram is constructed, the
random sample of \( R \) can be utilized to estimate the cardinality of \( R \bowtie S \) as with hash joins. This step enables cardinality estimation before sorting \( R \). In contrast, the \textit{dne} [33] and the estimator of Luo et al. [92] conduct cardinality refinement while the join is being processed. In addition, since a sorted input stream is necessarily clustered, their estimates fluctuate in the case of skewed data.

The above discussion assumed that the relations are not already sorted. If the relations are already sorted, it is not possible to push down estimation to a preprocessing phase. In these cases however, typically there is an index on the join column for one or both of the relations. This index can be utilized for cardinality estimation as illustrated for Index Nested-Loops joins in Section 3.3.4

### 3.3.3 Nested-Loops Joins

In hash joins and sort-merge joins, both inputs are read completely before any significant join processing is done. By building a histogram on one of the input relations, we are able to simulate the join in the preprocessing phase itself and obtain an estimate that converges progressively to the exact value. In the case of a nested-loops join, there is no such preprocessing of the outer input; it is joined as it is read in. Assume a nested-loops join of two relations \( R \) (outer) and \( S \) (inner). For each tuple (or block of tuples) of \( R \), the entire relation \( S \) is scanned. Therefore, a histogram can be constructed on the join attribute \( S.a \) during this scanning phase. This histogram can be utilized in conjunction with a random sample of \( R \) in order to obtain an accurate cardinality estimate of \( R \bowtie S \) at runtime.

In practice, nested-loops joins are often optimized by building temporary indices on the inner input (in the absence of a permanent index), and by pre-sorting the outer input to improve locality in memory access. In the presence of such preprocessing phases, we can construct estimators similar to the incremental estimator for hash joins, and get early estimates. The particular order in which histograms are constructed would depend on the optimization techniques used.

### 3.3.4 Index Nested-Loops Joins

If the join is an index nested-loops join, cardinality estimates are obtained in the following manner. Suppose the index is on the inner relation \( S \). A random sample of the outer relation \( R \) is then joined with the inner relation \( S \) using this index. This result is then scaled up appropriately to obtain cardinality estimates for the join. Following this, regular join processing continues.

Such an index based cardinality estimation scheme can also be used for Merge Joins when one or both of the relations are sorted on some attribute, with an index present (Section 3.3.2). Therefore, before the merge join is executed, a random sample of one of the join relations is read, and joined with the index of the other relation as in the case of index nested-loops joins.
3.3. Joins

Figure 3.2: Join Pipelines

3.3.5 Join Pipelines

In the previous section, we discussed how estimators can be deployed for simple join operators. Our primary contribution was the idea of pushing down estimation to the preprocessing phases of an operator. We now consider the case of multiple join operators on a pipeline. A common construction is a pipeline of hash joins, each taking the output of a lower hash join as its probe input. An example of such a join pipeline is shown in Figure 3.2(c), with the probe input shown on the right and build input on the left.

Note that the cardinality estimation procedure illustrated for binary hash joins does not extend directly to hash join pipelines. This is because, if the probe input itself is the output of a hash join on the same attributes, the probe input cannot be considered random anymore. In Figure 3.2 (a), suppose the joins are implemented as hash joins with the output of the lower hash join (JL) being the probe input to the upper hash join (JU). The probe input (C) of the lower join is joined with the build input (B) in a partitionwise manner. Therefore, the output of the join is clustered on attribute \( x \) as well, and we cannot assume that it is a randomly ordered input to the upper join. A similar problem arises if the joins for Figure 3.2 (a) are implemented as sort-merge joins. The output of the lower join is sorted on attribute \( x \), and does not need to be sorted again by a sort operator before it is processed by the upper join. In this case too, the randomness assumption is violated. Handling such clustering of output of a hash or sort-merge join is necessary if we have a pipeline of joins on the same attribute.

We illustrate our cardinality estimation procedure using two possible hash join pipelines. The first scenario is if the join columns of the lower join are involved in the upper join as well. We refer to such join pipelines as having joins on the same attribute. We refer to settings where the join
columns for the joins in the pipeline are different as *joins on different attributes*. Note that this definition does not have anything to do with the names of the joining columns. If the lower join in Figure 3.2 (a) had the condition \( B.x = C.y \), and the upper join had the condition \( A.z = B.x \), it would be still considered a case of joins on the *same* attribute since the join column of the lower join is involved in the upper join as well. For ease of exposition we present our handling of both kinds of pipelines of two hash joins (illustrated in Figure 3.2) (a) and (b), before we present the general case.

**Hash Joins On The Same Attribute**

In this setting we have a pipeline with joins on the same attribute (Figure 3.2(a)). It is possible to push the cardinality estimation for the upper join down to the lower join. If the joins in Figure 3.2 (a) are implemented as hash joins, \( A \) is the build input for the upper hash join while \( B \) is the build input for the lower hash join. We maintain counts \( N^A_v \) and \( N^B_v \) for each value \( v \) seen in the build phase for both joins. As the probe input is read, each tuple with value \( v \) in the join attribute will contribute \( N^A_v \cdot N^B_v \) tuples to the output of the upper join. Thus the cardinality estimate for the upper join can be refined in an online fashion as:

\[
D_{n+1} = \frac{D_n n + N^A_v N^B_v |C|}{n + 1}
\]

By the end of the first pass of the lower join, the estimator gives the correct estimate of the cardinality at the upper join as well. Such *push-down* estimation generalizes to any number of hash joins on the same join attribute. The central idea is to extract the value of the join attribute from the probe input at the lowermost join and to probe all the histograms of the build inputs in the pipeline with this value as illustrated in Figure 3.2 (a). Multiplying the counts returned by each of these histograms gives us the number of tuples of output that the given probe input will produce.

**Hash Joins on Different Attributes**

In this setting too, it is possible to push down cardinality estimation of all joins to the lowest join in the join tree. This procedure enables accurate cardinality estimates for all joins in the pipeline at the end of the first pass on the probe input at the lowermost join.

Consider the join tree shown in Figure 3.2 (a). If the joins were on different attributes, the join condition on the upper join \( JU \) could be either of the form \( A.y = C.y \) or \( A.y = B.y \), i.e., the join attribute of the probe input of the upper join could be an attribute of either the probe input or the build input of the lower join. We now describe these cases in detail.

- **Case 1** *Outermost probe input involved in all joins* (\( A.y = C.y \)): Our estimation procedures build histograms on the upper and lower build inputs to the joins on their join columns. With histograms on \( A.y \), and \( B.x \) built during the build phase, the probe input \( C \) is scanned. Now,
any tuple of \(C\) with column values \((x = x_1, y = y_1)\) will produce \(N_{y_1}^A N_{x_1}^B\) tuples in the output of the upper join. Since this information is readily available in the histograms, estimation can be performed as before, obtaining accurate cardinality estimates by the end of the first pass on the probe input of the lower join.

• **Case 2** *Outermost probe input not involved in all joins* \((A.y = B.y)\): Consider the pipeline in Figure 3.2(b). This case cannot be handled in the same manner as above. If histograms are built on columns \(A.y\) and \(B.x\) at the upper and lower joins, we would not be able to probe the upper histogram using the probe input of the lower join since there is no column \(C.y\). To get around this problem, we use the fact that it is possible to simulate the join of relations \(A\) and \(B\) on column \(y\) while building the hash partition for \(B\). Suppose we build a histogram on \(A.y\) while scanning relation \(A\). When relation \(B\) is scanned, for each tuple with values \((x = x_1, y = y_1)\), we update histogram on the \(x\) attribute of \(B\), incrementing the count of the bucket corresponding to the value \(x_1\) by 1. Additionally, another histogram representing the distribution of values in column \(x\) of \(A \bowtie_y B\) is maintained. The bucket of this histogram corresponding to \(x_1\) is incremented by \(N_{y_1}^A\). As a result, at the end of the build pass of relation \(B\), we have two histograms representing the distribution of \(x\); one on relation \(B\), and the other on \(A \bowtie_y B\). When the probe input to the lower join is read, both these histograms can be probed with the value of the attribute \(C.x\) to obtain an estimate of the cardinality of the output of the upper \(A \bowtie_y (B \bowtie_x C)\) and lower \(B \bowtie_x C\) joins as before.

The two cases described above provide the basic intuition on how to handle a pipeline containing a chain of multiple hash joins. Algorithm 1 describes the procedure in more detail. With each hash join operator, a structure *Node* is associated. Each *Node* contains a list of histograms *Node.histList* which is a linked list with pointers to histograms in the subtree below the node that are relevant for cardinality estimation at the node. The other important field is *Node.joinList*, which is a list of joins further up the pipeline that involve the build input to the node. Other fields in this structure are self explanatory. Algorithm 1 describes the estimation and histogram maintenance procedure as *getNext()* calls are made on the nodes of the join pipeline. We do not present the pseudocode for the *buildHistograms()* and *probe()* procedures for clarity, but describe their functionality in the explanation that follows.

A hash join node is initially in state *INIT*. On the first *getNext()* call to the node, it initializes its histogram and labels it with the \((Relation,Attribute)\) pair which is to be used to probe it. It also checks if its build relation is involved in a join further up the join tree. This is done by the function *makeJoinList()* which checks the labels of the histograms at nodes in the tree above. If it finds a histogram in the *joinList* of a parent node with a \((Relation,Attribute)\) label that matches its build input, it adds an entry for the \((Parent,Attribute)\) pair to the *joinList* of the node. This list contains all the histograms which have to be constructed as per Case 2 above. The node then
Algorithm 1 Estimation for hash join pipelines

```plaintext
proc getNext(Node node)
  if node.state == INIT then
    node.histList.add(node.probeRelation,node.joinAttribute)
    if type(node.parent) == hashjoin then
      makeJoinList(node,node.parent)
    end if
    node.state = BUILD
  end if
  if node.state == BUILD then
    while (tuple = getNext(node.buildChild))!=NULL do
      buildHistograms(node,tuple)
    end while
    node.state = PROBE
  end if
  if node.state == PROBE then
    tuple = getNext(node.probeChild)
    if type(node.probeChild) != hashjoin then
      probe(node,tuple)
    end if
  end if

proc makeJoinList(Node node,Node Parent)
  for all attribute ∈ node.buildRelation do
    if Parent.histList[node.buildRelation,attribute]!=NULL then
      node.joinList.add(Parent,attribute)
    end if
  end for
  Parent.histList.add(node.probeRelation,node.joinAttribute)
  if type(Parent.parent)==hashjoin then
    makeJoinList(node,Parent.parent)
  end if
```

moves to the BUILD state in which it reads in its build input and partitions it. In this phase, it builds a histogram on the build input, and also constructs a histogram for all entries in the joinList data structure. This functionality is provided by the call to buildHistograms(). Finally, the node moves into the PROBE state and reads in its probe input. We push down all estimation to the
lowermost node in the join tree; the \texttt{probe()} call is made only at the lowermost node. This function contains the code for incrementally obtaining estimates for node cardinality.

The following example runs through the algorithm for a 4-way join pipeline.

\textbf{Example 2.} We describe the execution of our algorithm on the join pipeline shown in Figure 3.2 (c). When the first \texttt{getNext()} call is made to the upper join node \texttt{JU}, it builds a histogram on \texttt{A.x}, and labels it as \((C, x)\) since \(C\) is the probe relation for the join. It then makes a \texttt{getNext()} call on its probe child \texttt{JM}. At this node, a histogram on \texttt{B.y} is initialized with label \((D, y)\). The \texttt{makeJoinList()} call does not find any matching histogram at its parent node, and so it just adds a pointer to the \((D, y)\) histogram to \texttt{JU.histList}. Finally we reach the lowermost join node \texttt{JL} where a histogram is initialized on \texttt{C.z} and labeled as \((D, z)\). The \texttt{makeJoinList()} call detects that there is a histogram at \texttt{JU} with a label \((C, x)\) corresponding to \(C\), and so it adds \((JU, x)\) to the \texttt{joinList} of \texttt{JL}. When \texttt{buildHistograms()} is called, it builds a histogram on \((A \bowtie_x C).y\) with label \((D, y)\) at the top node (Case 2), in addition to building the usual histogram on \texttt{C.y} at the bottom node. Now the probe input at the lowermost join is read. At this point each node contains appropriate histograms labeled by the columns of the probe input which are to be used to probe it and we can get estimates as in Case 1.

The algorithm as described here incrementally constructs and maintains lists of histograms required for cardinality estimation. To do so, it explicitly maintains information as to which relations join, and the attributes on which they join. Histograms are probed only if their label corresponds to the input being processed. New histograms corresponding to the distribution of join outputs are constructed if the input being processed is not the outermost probe input of the join pipeline.

\textbf{Other Join Algorithms}

For the case of sort-merge joins related issues arise. In particular a sequence of sort-merge joins on the same attribute can be handled in exactly the same way as a pipeline of hash joins. When the first relation is sorted, we can build a histogram on the sort column. Subsequent sorts build histograms on their own value frequency while probing corresponding histogram buckets to incrementally obtain the join estimate. For the case of a sequence of sort-merge joins on different attributes, we no longer have a pipeline. This is due to the fact that there must be a blocking sort operator between the output of a lower join and an upper join if they are on different attributes. As a result, in this case, our discussion on binary sort-merge joins readily applies.

Estimation in the case of a pipeline of nested-loops joins can be performed in an analogous fashion to hash join pipelines, with histograms constructed on the inner inputs. For index nested-loops join pipelines however, we exploit the existing indices for cardinality estimation, as detailed in Section 3.3.4.
3.3.6 Approximation techniques for histograms

The primary contribution of our cardinality estimation algorithms for joins is the use of histograms constructed at blocking points during query execution. By using these statistical summaries in conjunction with a random sample of the outer relation, we are able to obtain accurate cardinality estimates before any significant join computation has taken place. In the description of our algorithms for histogram construction, we restricted ourselves to exact histograms which store the frequency of every distinct value in the join column. As our experimental evaluation in Section 3.7 shows, the overhead of maintaining this distribution is low on modern systems with large memories. Our techniques would still work with any approximate statistical summary structure as well, with some loss of accuracy though. A natural question to ask is whether we can analytically and empirically quantify the tradeoffs associated with using approximate summary structures. In this section we present an analysis of the approximate summary structure construction problem for online cardinality estimation.

Optimal histograms

The problem of constructing the optimal histogram for various kinds of queries in different settings is a well studied problem in database literature [75]. In particular, the class of v-optimal histograms [78] has been proposed as the optimal class of histograms where the set of values in the distribution is not accurately maintained. The v-optimal error measure is also known as the Sum Squared Error measure [79], and it aims to minimize the variance of values within each bucket. Dynamic programming based algorithms have been proposed for the class of v-optimal histograms [79].

However, the optimal histogram for our setting is not necessarily the v-optimal histogram. We require a single relation histogram which minimizes the error in the join result size, which is a function of 2 or more relations. This is in contrast to typical histogram construction problems which are formulated over a single relation, with a specified budget and an error function to be optimized that is defined on that single relation. We now describe an algorithm for constructing the optimal histogram for this case, and illustrate the challenges due to which this technique is not suitable for our purposes.

Consider a hash join between two relations $A$ and $B$. Let $A$ be the build input and $B$ be the probe input. Let the join be on column $x$ with the domain $V = x_1 \ldots x_n$, where $x_i < x_j$ for $i < j$. The frequency of $x_i$ in $A.x$ is $N_i^A$ and in $B.x$ is $N_i^B$. Thus the size of the join is $\sum_i N_i^A N_i^B$.

The histogram of an attribute $A.x$ is constructed by partitioning the domain $V$ into $k$ buckets, and approximating the frequencies inside each bucket by taking their average. Let the estimate for the frequency of value $x_i$ in relation $A$ given by a histogram be $H_i^A$. Our goal here is to estimate the cardinality of the join between $A$ and $B$. Therefore, we use as an error function, a variant of
the v-optimal error measure:

$$JE = \sum_i (N_i^A N_i^B - H_i^A N_i^B)^2$$

$JE$ refers to the Join Error i.e. the error in the estimate of the join cardinality due to the histogram. We use this definition of error since it is monotonically increasing. Note that we use the histogram only to approximate the inner relation $A$. This in accordance with our estimation procedure defined in Section 3.3.1 where histograms are built only on the inner/build relation. Given this setting, the optimal histogram construction problem is to construct a histogram that minimizes the error function.

Let $JE(i,k)$ be the minimal join error over the restricted domain $V(1,i) : x_1 \ldots x_i$ when $k$ buckets have been allocated to the domain $V(1,i)$. Let $E(i,j)$ be the join error induced by allocating a single bucket to the range $V(i,j)$. Then, if $F(i,j)^A$ is the average frequency for attribute $A.x$ for the range $x_i \ldots x_j$, we have

$$E(i,j) = \sum_{i \leq l \leq j} (N_i^A N_l^B - F(i,j)^A N_l^B)^2$$

Therefore, we have

$$JE(i,k) = \min_{1 \leq j < i} [JE(j,k-1) + E(j+1,i)]$$

This relation can be used to define a simple dynamic programming algorithm to compute the optimal histogram for our purposes. Given a budget of $k$ buckets, we can find $JE(n,k)$ using the dynamic program.

However, there are various problems with this formulation that make it unsuitable for our purpose. In particular, this formulation does not lead to any space savings since we have to store the full distribution on column $A.x$ while the optimal histogram is constructed. What we really need is a single pass algorithm that constructs the histogram on $A.x$ while the hash table is being built. However, the optimal histogram for $A.x$ depends on the distribution of values in the joining column $B.x$. Given that we have strict requirements on the order in which relations are seen, we must construct an approximate statistical summary on $A.x$ without seeing the distribution on the joining column $B.x$. These concerns illustrate the challenge in defining the optimal histograms for query monitoring. We now define a class of statistical summaries that can be constructed in one pass, and which enable accurate cardinality estimation for the purposes of query monitoring.

**Single pass statistical summaries**

We now present our technique for computing a summary structure that captures the data distribution of the inner relation to enable cardinality estimation. Our technique has low memory requirements and does not require us to store the entire data distribution. Before we describe our algorithms, we outline the requirements that a summary structure must satisfy:
• **Membership queries:** A summary structure should answer membership queries with a high degree of accuracy. Given a tuple of the outer relation, the structure should return a boolean value signifying the presence or absence of a matching inner tuple. We would like the error rate of the data structure to be low. Bloom filters [16] are a well known solution to this problem. In addition, they have also been previously used in conjunction with hash joins for early filtering and cache conscious hashtable management [35]. We adopt bloom filters in our solution.

• **Frequency queries:** If the outer tuple has (one or more) matching inner tuples, the summary structure should also be able to return the number of matching inner tuples, with a low error rate. If the value set for the column is maintained accurately, then the class of serial histograms [74] is known to be optimal for such matching problems. However, since we do not maintain a list of values, we would like to ensure that at least the high frequency values are known correctly along with their frequencies. This is the class of end-biased histograms. In our setting, we wish to extract the high frequency values from a data stream. This is known as the heavy hitters problem in data streaming literature, and has been widely studied [42, 96, 84, 39]. Each of these techniques has associated tradeoffs and varying ease of implementation. Since our techniques are to operate during query processing, our goal is to select lightweight techniques that can easily be pushed inside the database engine. To this end, we select the multistage filters technique introduced in [42], which is very similar to bloom filters. Despite this technique not having the best possible theoretical guarantees, we adopt this technique since it enables us to combine processing with bloom filters.

We utilize bloom filters in conjunction with multistage filters to get a summary structure that answers membership and frequency queries for the join column of the build relation. We now provide a brief overview of these techniques before describing our overall algorithm.

**Bloom Filters:** Bloom filters are a space efficient data structure for answering membership queries on a set. In particular, they have the property that they do not allow false negatives, while having a tunable false positive rate. A bloom filter is a bit vector $B$ of $m$ bits, all initially set to 0. In addition, it uses $k$ independent hash functions $h_1(x), \ldots, h_k(x)$ that map a value to a position in the bit vector. Each value of the set is hashed using the $k$ functions, and each of the $k$ bits is set to 1. Thus, for each inner tuple, we apply the $k$ hash functions on the value of its join attribute to set $k$ positions in the bit array.

Membership queries on a bloom filter are performed similarly by hashing the value with the $k$ hash functions and checking the corresponding bit positions. If all the $k$ positions have bits set to 1, then the bloom filter returns $True$, else it returns $False$. In our setting, we hash the outer tuple on its join attribute, and use the bloom filter to check if it has a matching inner tuple.
After $n$ elements have been seen by a bloom filter, the probability that a specific bit is set is:

$$p = (1 - \frac{1}{m})^{kn}$$

This implies that the probability of a false positive is $(1 - p)^k$ i.e. the probability that given $k$ bits, all are set to 1. In addition, bloom filters can also be used to approximate the number of distinct values in the inner relation. If any of the $k$ positions to which an inner tuple is hashed are set to 0, we increment a counter. This is a lower bound on the number of distinct values in the join column of the inner relation. Typically bloom filters are specified in terms of the number of bits per value of the set, and the number of hash functions. For example, a bloom filter with 8 bits per value and 3 hash functions has a false positive rate of approximately 2%. We now describe multistage filters which we use for frequency queries

**Multistage filters:** Introduced in [42], multistage filters are a technique to identify large flows in network traffic. Like bloom filters, they utilize $k$ independent hash functions. Each hash function is associated with an array of counters, which are all initialized to 0. Figure 3.3 describes the multistage algorithm with 3 hash functions. Each value is hashed using hash function $h_i(v)$ to array $A_i$ and the counter $A_i[h_i(v)]$ is incremented. If the counter exceeds the threshold $T$ for each array $i$, the value is identified as a heavy hitter, and its count is explicitly maintained separately. Because heavy hitters are identified only if each counter is above the threshold $T$, the probability of false positives is low. Like bloom filters, multistage filters guarantee no false negatives.

![Figure 3.3: Multistage Filters Algorithm](image)

Estan and Varghese show that if we have $b$ buckets, $k$ hash functions and $n$ distinct values, with $C$ elements in the relation being filtered, then if $d = \frac{Tb}{C}$, we have the following Lemma

**Lemma 1.** The probability that a value with frequency $f \leq T(1 - 1/d)$ passes a multistage filter is at most $p_f \leq (\frac{1}{d} \frac{T}{T-f})^k$

This bound is independent of the distribution of frequencies and order of the stream and is shown to be conservative for zipfian distributions. Estan and Varghese also prove bounds on the error rates in the returned frequencies of the multistage filter, and memory requirements of the filter. Further details may be obtained in the original paper [42].
Algorithm 2 Updating and Probing Bloom and Multistage filters

\textit{UpdateFilters}(Value \( v \))
\begin{verbatim}
new = false
heavyhitter = true
numvals++
for all \( h_i \in \text{hashfunctions} \) do
    hashval = \( h_i(v) \)
    bit = hashval\%bitarraysize
    if bitarray[bit] == 0 then
        new = true
        bitarray[bit] = 1
    end if
    index = hashval\%arraysize
    \( A_i[index]++ \)
    if \( A_i[index] \leq T \) then
        heavyhitter = false
    end if
end for
if new == true then
    distinctvals ++
end if
if heavy = true then
    UpdateCounters(v)
end if
UpdateAvg(v)
\end{verbatim}

\textit{ProbeFilters}(Value \( v \))
\begin{verbatim}
if checkBloomFilters(v) == false then
    return 0
else if checkHeavyHitters(v) == false then
    return Avg
else
    return Counter(v)
end if
\end{verbatim}
3.3. Joins

The Combined Algorithm:

Algorithm 2 describes our algorithm for updating and maintaining a combined summary structure that uses a bloom filter and a multistage filter. When the inner relation is being built, the function \( \text{UpdateFilters}(v) \) is called with the value \( v \) on the join column. The function updates a bloom filter and multistage filter using a set of hash functions \( h_i() \). In addition, the function also checks if the value is new, using the bloom filter, and whether it is a probable heavy hitter using the multistage filter. If the value is new, the set of distinct values is updated. Likewise, if the value is a heavy hitter, its count is maintained separately. Finally, the average frequency of the low frequency (non heavy hitter) elements is updated.

When the outer relation is read in, the function \( \text{ProbeFilters}() \) is called for each tuple to find the number of matching elements in the inner relation. \( \text{ProbeFilters} \) first checks the bloom filter to check for membership of the value in the inner relation, and if found, checks the list of heavy hitters. If the value is a heavy hitter, it returns the stored frequency, otherwise it returns the average frequency for the remaining elements.

The functions defined in Algorithm 2 are invoked Algorithm 1 in the following manner. The call to \( \text{BuildHistograms} \) in Algorithm 1 builds all the statistical summary structures required at a join node. Each such function call involves multiple invocations of \( \text{UpdateFilters} \) from Algorithm 2 to update the associated structures. Likewise, the \( \text{Probe} \) call in Algorithm 1 involves a call to \( \text{ProbeFilters} \) in Algorithm 2.

There remains one final issue to be resolved; We need to initialize the bloom filters and multistage filters appropriately. A bloom filter requires initialization of a bit array to a certain size, which is given by the size of the set multiplied by the number of bits per tuple. This set is the size of the domain from which values are drawn to be hashed and probed. We approximate this set size by setting it to the estimated size of the inner relation, with a minimum threshold to ensure that the bloom filter is not too small. Similarly, a multistage filter requires setting a threshold, which is specified as a fraction of the size of the inner relation. In both cases, we require an estimate of the size of the inner relation being hashed before these filters can be initialized appropriately.

Getting the size of the inner relation is straightforward if there are no selection conditions on it. In the presence of selection conditions however, the problem becomes more challenging. To obtain an estimate at a low overhead, we utilize a random sample of the base table of the relation, and pass it through the selection conditions to get an estimate of the selectivity of the conditions. This is then scaled up to the size of the base table to get an estimate for the size of the relation. This estimation procedure is very similar to the case of estimation in join pipelines since a selection predicate can be treated as a matching condition to a list that contains the set of values satisfying the predicate. For example suppose the estimate of the size of the relation \( S \) passing through the selectivity predicate \( \phi \) using a sample of size \( n \) tuples is \( D_n \). We then have:
\[ D_{n+1} = D_n \cdot n + I_\phi(s_{n+1}) \cdot |S| \\
\]
where \( I_\phi(s_{n+1}) \) is 1 if tuple \( s_{n+1} \) satisfies the predicate \( \phi \), and 0 otherwise.

Finally, we note that if the inner input is the result of a join (as in a left-deep or bushy plan), we can utilize the join cardinality estimation techniques introduced previously to obtain an estimate of the size of the input. Our techniques ensure that accurate cardinality estimates are obtained before significant join processing is performed. These estimates are produced before the join starts producing tuples. Therefore, we can use these cardinality estimates to set the size of bloom filters and multistage filters.

A similar issue arises when we build histograms on the result of the join of two relations that are both inner inputs to the pipeline. Consider for instance the pipeline shown in Figure 3.2 (b). To get an estimate of the size of \((A \bowtie B)\), we utilize a random sample of the lower relation \(B\), and simulate the join of it with the histogram on \(A.y\). This is used to set the size of the bloom filters and multistage filters for the approximate summary structure on column \((A \bowtie B).x\).

As an example of the memory requirements of this procedure, suppose the inner relation has 100 K tuples. A bloom filter with 8 bits per tuple and 3 hash functions will then take 100KB space in memory. Similarly, a multistage filter with the same 3 hash functions and arrays of size 1000 elements will take 12KB of memory. A threshold \( T = 100 \) can be set to catch the high frequency values while maintaining the average frequency for the remaining values. In our experiments, we parameterize multistage filters using a parameter \( F \), which is the ratio of the size of the relation and the Threshold \( T \).

### 3.4 Aggregation

In database systems, aggregation is typically implemented by sorting or hashing. In a hash based aggregation, the input is read and partitioned using a hash function and then data is grouped by reading in the partitions. In sort-based aggregation, the input is first sorted on the group-by attribute and aggregates are computed by reading in the sorted input. In both cases one can obtain accurate cardinality estimates of the output by counting the number of groups seen in the hashing/sorting phase. Doing this requires maintaining a data structure for storing the set of grouping values that have been seen.

Although counting the number of groups seen in the preprocessing phase provides the cardinality of the output of the aggregation operator, we would like to be able to estimate this cardinality even before the entire input is observed. This is the distinct value estimation problem and it has been well studied in the database and statistics literature (see [60, 25, 62] and references therein). However, previous work on distinct value estimation has been conducted in an offline setting, which admits techniques with a higher runtime overhead. For selectivity estimation used in query
optimization, this makes sense since optimization is a one-time operation and high accuracy is the prime objective. For progress estimation however, the overheads of obtaining estimates are of equal importance. During progress estimation one has the ability to examine progressively more tuples and refine the estimate. As a result, accurate but low overhead estimation techniques for distinct values are important.

In the literature, the state of the art estimator for the number of distinct values is the GEE (Guaranteed Error Estimator) [25]. Let $T$ be a random stream of attribute values of size $|T|$. GEE has guaranteed accuracy as it can provide a lower and upper bound guarantee for its estimate. It operates based on the principle that data can be split into low-frequency and high-frequency groups and that any reasonably sized sample should contain all the high frequency groups. It treats the singleton values in the sample as a sample of the low frequency values and scales them up to minimize the error. If $f_i$ is the number of groups that occur exactly $i$ times in the sample of size $n$ from $T$, the GEE estimate for the number of distinct values in a relation of size $|T|$ is

$$D_n = \sqrt{\frac{|T|}{n}} f_1 + \sum_{j=2}^{n} f_j$$

The estimate follows by taking the geometric mean of the lower and upper bound estimate for the number of singleton groups ($f_1$ and $\frac{|T|}{n} f_1$ respectively). Assuming that we accumulate the frequencies with which values appear (i.e., construct a histogram of the attribute value frequencies observed in the sample of $n$ values of $T$) we can compute the GEE estimate by a single pass on the histogram. However, we observe, that it is also possible to compute this estimate by incrementally maintaining the histogram as $t$ increases in size. Consider what happens when a new value $i$ is seen. If $i$ has never been seen before, it contributes to the first term in the GEE estimate. If $i$ has been seen before it contributes to the second term. If $i$ had been seen only once before, its contribution to the first term must be deducted. If $S_1$ and $S_m$ represent the number of groups seen once and more than once respectively, the estimator can be updated using Algorithm 3.

As the original paper on GEE attests, the estimator works best for data with high skew. However, it tends to overestimate the number of groups when the sample size is small. It also doesn’t perform well on low skew data that have a large number of distinct groups. For this reason, a heuristic is proposed to obtain analytically an estimate for the number of singleton groups. There are two problems with this approach: (a) obtaining the analytical estimate involves solution of equations using numerical methods, which makes the technique unsuitable for adoption inside a query processing engine and (b) the problem still persists, because like singleton groups, groups of size two or three may constitute a large fraction of the input. In such cases the estimate will continue to be highly inaccurate.

We observe that the cases in which the GEE estimator fails are exactly the cases in which a large number of groups appear in the input having relatively low variance in their frequencies. Our goal is to obtain an improved estimate in the cases where GEE fails, but at the same time keep
### Algorithm 3 Updating the GEE Estimator

```python
while getNext(T) do
    $i =$ value of the new tuple on its grouping attribute
    if $N_i == 0$ then
        $S_1 = S_1 + 1$
    else if $N_i == 1$ then
        $S_1 = S_1 - 1$
        $S_m = S_m + 1$
    end if
    $N_i = N_i + 1$
    $n = n + 1$
    $D_n = \sqrt{\frac{|T|}{n}}S_1 + S_m$
end while
```

the runtime overhead of obtaining such an estimate low to make it suitable for progress estimation. To this end, we use a maximum likelihood estimator (MLE) which is known to have bounded variance [17]. We use the estimator defined in Section 2.2.3 written below:

$$D_n - \hat{g} = \sum_{i=1}^{n} f_i (1 - \frac{i}{n})^n - \sum_{i=1}^{n} f_i (1 - \frac{i}{n})^{|T|}$$

where $t$ is the sample size currently read, and $M$ is the maximum frequency count for any group as currently observed. This estimator rarely overestimates the number of groups, although it is prone to underestimation. Unlike the GEE estimator, as will be demonstrated in Section 3.7.1, it works best when the data has low skew.

The MLE estimator cannot be incrementally maintained like the GEE estimator and so it must be recomputed regularly. Setting a constant interval for recomputing the estimate is not a good idea since we would like to refine our estimates more often when they are changing frequently. Algorithm 4 presents our method for refining our estimates. We define a lower ($l$) and upper ($u$) threshold for the interval of recomputation ($I$) and start off recomputing the estimate every $l$ tuples. If the new estimate is within $k\%$ of the old estimate, we double our interval, provided it is less than $u$; otherwise we set our interval back to $l$. This method ensures that we recompute the estimate more frequently when it is needed.

The GEE estimator is inexpensive, but it can also cause severe overestimates for small sample sizes. It works best for data with high skew. The MLE estimator is somewhat more expensive but works best when data has low skew. It is natural to make a choice between the two based on the skew of the data and the cost of computing the estimate.

In order to get a measure of skew, we compute the squared coefficient of variation $\gamma^2$ of the frequencies of the groups seen. The quantity $\gamma^2$ is a measure of the variance of the frequencies
3.4. Aggregation

Algorithm 4 Interval setting for MLE

getMLE(): function to compute the MLE estimate

\[ I = l \]

while getNext(T) do

\[ n = n + 1 \]

if \((n \mod I) == 0\) then

\[ E_{old} = E \]

\[ E = \text{getMLE}() \]

if \(1 - k < E_{old}/E < 1 + k\) then

if \(I < u/2\) then

\[ I = I \times 2 \]

end if

else

\[ I = l \]

end if

end if

end while

of the groups, and a low value reflects that the data has low skew. In addition, it is inexpensive to compute since it can be incrementally maintained as new tuples are seen. This is achieved by decomposing the coefficient of variation formula to elements – prefix sums and prefix sums of squares – that can be maintained incrementally. We set a threshold \(\tau\), and use the MLE estimator if \(\gamma^2 < \tau\), and the GEE estimator otherwise. (See Section 3.7.1 for experiment details)

Our estimation procedures for aggregation assume access to a random sample of the relation (or join of relations) on which a group-by is being performed. This is typically available only if the aggregation is on a base table, or on top of a materialization point. In the absence of such a random sample, estimation is still performed on the input stream of tuples. This is because our techniques are guaranteed to eventually converge to the correct answer when the entire relation has been read. Moreover, since we push down our estimation into the sorting or hashing step preceding a group-by, we are guaranteed accurate cardinality estimates before the aggregation operator starts processing tuples. In the particular case where there is no such preprocessing step such as a sort-merge join producing tuples for a sort-based aggregation operator on the same attribute. In this case, it is possible to push down the estimation for the aggregation into the preprocessing phase of the join. In addition to computing the estimate of the cardinality of the output of the join, we also build a histogram storing the frequency distribution of the output. The GEE or MLE estimators can now be run using this histogram as input in order to estimate the cardinality of the output of the aggregation operator.
3.5 Progress Estimation

In Sections 3.3 and 3.4 we introduced a set of techniques which enable refinement of cardinality estimates at query execution time. Our techniques hinge on the fact that operators for joins and aggregations often have preprocessing phases where the entire input is observed and partitioned typically by sorting or hashing. We have shown that it is possible to push down cardinality estimation for operators into these preprocessing phases. Pushing down estimation enables accurate cardinality estimation before the operator has performed significant processing. We revert to the dne estimator for operators that do not have any preprocessing phase, such as index nested-loop joins and selections. Since we assume that part of the input is seen in random order, the dne estimator has zero error in expectation [29], and so we use the dne estimator to handle these cases.

A query \( Q \) consists of multiple pipelines. If we label the pipelines as \( (p_1 \ldots p_n) \), the \( gnm \) estimate of progress is

\[
gnm = \frac{C(Q)}{T(Q)} = \frac{\sum_{i \in p_1} K_i + \ldots + \sum_{i \in p_n} K_i}{\sum_{i \in p_1} N_i + \ldots + \sum_{i \in p_n} N_i}
\]

For a pipeline \( p \), let \( C(p) \) be the number of \texttt{getnext} calls made over all operators in it i.e. \( C(p) = \sum_{i \in p} K_i \). Similarly let \( T(p) \) be our estimate of the number of \texttt{getnext} calls that will be made over the run of the pipeline i.e. \( T(p) = \sum_{i \in p} N_i \). Of these pipelines, some would have already finished executing. For these, we know \( C(p) \) and \( T(p) \) precisely. For pipelines, that are currently executing, we know \( C(p) \) precisely, while we estimate \( T(p) \) using our estimation techniques. For pipelines that are yet to begin, \( C(p) = 0 \), and we refine the optimizer estimates for \( T(p) \) using upper and lower bounds as in [33]. Summing these values over all pipelines gives us \( C(Q) \) and \( T(Q) \), and we estimate the progress of the query as \( C(Q)/T(Q) \).

**Example 3.** Consider a query executed as a hash join of two relations \( R \) and \( S \) with \( S \) being the inner relation. The query evaluation plan consists of 2 pipelines. The first pipeline consists of a table scan of \( S \), with selection predicates applied, which is then stored as a hashtable. When this pipeline is executed, we estimate the selectivity of the predicates on \( S \) by utilizing a random sample of \( S \), applying the predicates, and scaling up the result size. The second pipeline consists of the table scan of \( R \), and the join operator. When this pipeline is executed, we estimate the cardinality of the join using a random sample of \( R \), and the histograms constructed on \( S \).

Our techniques are designed to perform cardinality estimation for only the currently executing join pipeline. Estimates for future pipelines are refined using only lowerbound-upperbound refinement techniques such as those introduced in [33]. We note that this inability to estimate beyond the currently executing pipeline is a consequence of the requirement that we do not access relations out of order i.e before the query evaluation plan accesses them.
We clarify here again that our techniques require only the presence of a random sample of certain inputs to the pipeline. Essentially, our techniques utilize this random sample for cardinality estimation before processing the currently executing pipeline. We also note that all our techniques are guaranteed to eventually converge to the correct progress estimate, even in the absence of a random sample. This is similar to the guarantees associated with previous research on progress estimation [92, 93, 33, 29].

3.6 System Design and Implementation

In this section, we provide an overview of the design and implementation of our system as illustrated in Figure 3.4. The query optimizer generates a query execution plan which is then executed by the database engine. We introduce an additional plan analyzer component which extends the execution plan with additional code and data structures for the purposes of cardinality estimation. This extension is performed by adding additional wrapper code around each operator. The wrapper code can access the tuples flowing in and out of each operator and can access relevant estimation information stored in other operators through a global data structure. Additionally, the wrapper can access the frontend described in Section 3.8 through network sockets.

We instantiated this system design in the Postgresql 8.0 database engine. As a first step, we implemented the gnm model along with the dnc estimation framework [33]. This was achieved by modifying the central control function for query execution in PostgreSQL, which acts like a wrapper for all operators. We also changed the data structures for operators and added counters for the number of tuples emitted by the operator, the estimated cardinality of the operator, and upper and lower bounds for the cardinality.

We then implemented our proposed estimators within each operator. The modifications con-
sisted of adding calls to functions for histogram construction, random sample processing and cardinality estimation using the techniques mentioned in the previous sections. We also modified the table scan operators to access on demand a precomputed block-level random sample of the base tables. The random sample is utilized only for the purposes of obtaining an initial cardinality estimate. Following this, the scan operator is reset and the input is scanned again for regular query processing. This ensures that any constraints assumed by the query plan with respect to the order of tuples processed are not violated.

3.7 Evaluation

In this section, we evaluate the accuracy of the estimations introduced in the previous sections and also quantify the overheads of ConEx.

We evaluated the system described in Section 3.6 on a machine running Fedora Core 3 with a 2.80 GHz processor and 1 GB RAM. The system was executed with the network sockets interface to the frontend disabled in order to capture the costs of our estimation framework. We utilized a publicly available tool [32] to generate skewed data conforming to the TPC-H schema. We modified this tool in order to be able to vary the number of distinct values in a table column. In what follows, we first describe the experiments for evaluating the accuracy of our estimation methods, and then move on to measuring the overheads.

3.7.1 Evaluating Accuracy

We define the ratio error ($R$) of an estimator as the ratio of the estimated cardinality of the output of the operator to the final cardinality of the output. Our presentation of the accuracy of our estimators is in terms of the ratio error. The sooner the estimator reaches a ratio error of 1, the better it is. This definition of ratio error generalizes in a natural way to complete query plans. For a query, if $T'(Q)$ is the current estimate of the number of $\text{getnext}$ calls over the query, and $T(Q)$ is the actual number of $\text{getnext}$ calls that will be made, then as per gnm:

$$\frac{\text{Actual Progress}}{\text{Estimated Progress}} = \frac{C(Q)/T(Q)}{C(Q)/T'(Q)} = \frac{T'(Q)}{T(Q)} = R$$

Methodology

Our experiments on the accuracy of the estimators were all conducted on tables complying with the $\text{customer}$ and $\text{nation}$ schemas of the TPC-H specification and for ease of presentation we restricted them to just the $\text{nationkey}$ attribute of these tables. $\text{Nationkey}$ is a primary key of the $\text{nation}$ table, and a foreign key to the $\text{customer}$ table; it assumes values from $[1 \ldots 25]$ in the TPC-H specification. In order to flexibly to test the estimators on a variety of data sets and domain sizes, we modified the generating function for the relevant join columns. For the accuracy experiments
we present, we keep the number of rows of the table equal to that resulting by running TPC-H data generation with a scaling factor of 1.

In the discussion that follows, $C(z,n)$ denotes a customer table having a Zipfian distribution with skew $z$ ($Z = z$) on the nationkey column, taking values from the set $[1 \ldots n]$. Thus, for example $C_{2,125K}$ would denote a customer table with Zipfian skew of 2 on the nationkey column which takes values between 1 and 125000. To simulate join operations between columns in which key value frequency exhibits different correlation patterns, we generate skewed distributions in which the peak value frequency corresponds to different values. We use superscripts to distinguish between tables that have the same skew, but different distributions of frequencies across the domain of values. So $C_{2,125K}$, $C_{1,125K}$, $C_{2,125K}$ all have Zipfian skew of 2 on the nationkey column, but the values with a high frequency in one table may have a low frequency in another table. In a join between two columns, if the high frequency values in one column are also the high frequency values in the other column, then estimating the size of the join is easy since most of the output of the join is due to the join of these high frequency values [45, 31]. In most of the experiments, we try to capture distributions that have different high frequency values. This is the classic worst case scenario for join cardinality estimation, where low frequency elements of one table could possibly join with high frequency elements of the other table and vice versa. In these cases, the cardinality estimates of the optimizer are often inaccurate.

With this notation in place, we describe the results of the accuracy of the estimators. We concentrate on evaluating how quickly the estimators converge to correct cardinality estimates. We conduct the experiments by first obtaining a random sample of some percentage of the input (labelled “Probe Input Seen (%))” or “Probe Input Joined (%))” in the graphs). Unless stated otherwise, the experiments utilize exact histograms to capture the data distribution. We report accuracy results using approximate summary structures in Section 3.7.1.
Chapter 3. ConEx: Query Monitoring

Joins

We first present results for the estimation procedures for hash joins. The join algorithm used was the hybrid hash join as implemented in Postgresql. To evaluate the accuracy of our estimators for joins, we generated customer tables with different domain sizes and varying skew on the nationkey attribute. In each case, we evaluate joins between two tables with the same domain size and skew, but different distributions i.e a high frequency value in one table may be a low frequency value in the other.

We present results for the ratio error on small domain sizes (5K elements) in Figure 3.5(a), and large domain sizes (125K elements) in Figure 3.5(b). In these graphs, the value of $Z$ corresponds to the skew parameter of the Zipfian distribution. In each case two Zipfian distributions with the same skew and different peak frequency values are joined. In all cases, as is evident in the figures, our estimators converge to an approximately correct ratio error estimate while having seen only a fraction of the probe input. It is evident that only a small percentage sample of the input is required in each case, to accurately obtain estimates and trace progress precisely.

Figure 3.6 shows a comparison of our estimator (labeled as once for online cardinality estimation) with the dne estimator and the estimator of Luo et al. [92] (labeled as byte since their framework is based on counting bytes). Since the other two estimators see data as it is output by the join, the graphs are parameterized with respect to the percentage of the probe input that has been joined. In both figures, our estimator has already converged to the correct cardinality estimate when only a small percentage of the probe input has been joined. This is because the estimator has already observed the entire input during repartitioning of the probe input. Figure 3.6 (a) corresponds to a join between the customer tables $C_{1,125K}$ and $C_{1,125K}^1$ on the nationkey attribute. In this case, the PostgreSQL cardinality estimates are off by about a factor of 13. The byte estimator imposes a weighted average operation involving the original cardinality estimate, and so it converges slowly to the correct answer. The dne estimator, disregards the original optimizer estimate as soon as the pipeline starts executing. However, it underestimates the number of tuples to be produced by the operator because of the effect of reordering caused by the partitioning of the probe input as

![Comparison of estimators for binary hash joins](image)
3.7. Evaluation

![Figure 3.7: Estimation With large relations](image)

(a) Sort-Merge Join  
(b) Index Nested-Loops Join

Figure 3.7: Estimation With large relations

described in Section 3.3.1. Similarly, Figure 3.6 (b) corresponds to a primary-key foreign-key join between a customer table \( C_{(2,125K)} \) and its corresponding nation table with a selection condition \( \text{nationkey} < 50000 \). Even for this scenario, the byte and dne estimates are highly inaccurate and remain inaccurate until most of the probe input has been joined. The reason for this inaccuracy is repartitioning of the probe input, which for high frequency values implies that the joining values are concentrated in a hash few buckets. The purpose of Figure 3.6 is to stress on the importance of reordering caused by hash partitioning in a hash join. Similar effects are caused by sorting before a sort-merge join. Despite the fact that tables are retrieved in random order from disk, the dne and byte estimators do not converge smoothly and quickly to correct cardinality estimates because they perform estimation after the data has been hashed or sorted.

Other Join Algorithms: We now present experiments that demonstrate ConEx for other join algorithms. In these experiments, we utilized the Lineitem and Orders tables of the TPCH schema, and join them on the orderkey attribute. These tables have 6 million and 1.5 million tuples respectively. As before, we drop the primary key - foreign key constraint between the Lineitem and Orders tables, and generate skewed distributions for these columns with different peaks (high frequency values). We generate tables with varying skew on the join column, with the domain size set as 1.5 million. This is to capture the challenging case of cardinality estimation when a high frequency value of one table might join with a low frequency value of the other. These experiments additionally demonstrate that our techniques are scalable to larger domains as well.

Figure 3.7(a) shows the accuracy of our estimation procedures when executed with a sort-merge join algorithm on unsorted inputs. In this case, our techniques build a histogram on the join column of the inner relation that is sorted first. Then a random sample of the outer input is read, and utilized to estimate the cardinality of the join. The join columns are generated with Zipfian skew varying between 0 and 2. As can be seen, our estimation procedures converge to the correct join cardinalities. The relatively higher error for the \( Z = 2 \) case is a reflection of the difficulty of join cardinality estimation on highly skewed join columns. In this case, a low frequency value in one relation may join with a high frequency value in the other relation making
cardinality estimation difficult. We note however, that our estimation procedures ensure that accurate cardinality estimates are obtained before the merging step of the sort-merge join.

In Figure 3.7(b) we change the join algorithm to the Index Nested-Loops join algorithm. As before, we join Lineitem and Orders relations on the orderkey attribute. However, we keep the skew on the join column fixed to $Z = 1$. We compare our estimation procedures to the $\text{pmax}$ and $\text{safe}$ procedures introduced in [29]. The figure clearly shows the accuracy of our online cardinality estimation techniques.

The $\text{pmax}$ procedure estimates the number of getnext calls to be made on the join using a lowerbound estimate $LB$. In the case of a nested loops join, $LB$ is the number of getnext calls currently made to the join, which is equivalent to the number of tuples currently produced by the join. As a result, we can see that the $\text{pmax}$ procedure continuously increases its estimate for the join as more tuples are produced by the join algorithm.

The $\text{safe}$ procedure estimates the amount of work to be done by the join as $\sqrt{LB \times UB}$ where $LB$ and $UB$ are lower and upper bounds on the cardinality estimates of the join. In the case of a Nested-Loops Join on columns that are not primary keys, the upper bound is equivalent to the product of the size of the inner relation and the size of the remaining part of the outer relation. This is because each remaining unseen tuple of the outer relation could potentially join with every tuple of the inner relation. The current set of experiments were performed on the Lineitem and Orders tables with the join columns modified so that the primary key - foreign key constraint no longer holds. As a result, in the absence of other information, the $\text{safe}$ estimator significantly overestimates the work to be done by the join. However, we do note that the errors of the $\text{safe}$ estimator are much lower for primary key - foreign key joins, where the upper bound $UB$ is bounded by the size of the foreign key relation.

**Join pipelines**

We now present accuracy results for pipelines of joins. We present both settings namely the setting where the join column of a lower join is also the join column of the upper join, and the setting
where it is not as discussed in Section 3.3.5.

Figure 3.8 shows the behaviour of our estimator for the setting where the join column is the same. In this experiment, we use $C_{(z,5K)}$, $C^1_{(z,5K)}$ and $C^2_{(z,5K)}$ and present three curves one for each value of $z$ ($z = 0, 1, 2$). In each experiment the three relations joined are generated using the same $z$ value. Figure 3.8 (b) shows the estimate of the cardinality of the lower join as its probe input is seen while Figure 3.8 (a) shows the estimate of the cardinality of the upper join, also with respect to the percentage of the lower join seen. Notice that for the $Z = 2$ plot, the estimate of the cardinality of the upper join is inaccurate in between before converging to the correct value. This point corresponds to a value of the lower join, joining with a high frequency value of the upper join. The reason why it is so prominent is that in this case, only a few values actually contribute to the join.

Figure 3.9 shows the behaviour of our estimator in a two join pipeline where the join columns of the upper and lower joins are different. The join attribute of the probe input of the upper join can come from either the probe input of the lower join or the build input of the lower join (Case 1 and Case 2 respectively in Section 3.3.5). Here too we want to recreate the worst case scenario for joins as described in Section 3.7.1, and this time for two different join columns. To do so, we replace the primary key column $custkey$ for the customer relation with a skewed distribution on a domain with 25K elements. We also set the domain of the $nationkey$ column to 25K elements. The lower join is kept fixed between $nationkey$ columns with the same skew, and the upper join is between $custkey$ columns with varying skew. All join conditions are on columns which have the same skew, but different distribution of frequencies across the domain of values. We describe a subset of our experiments here.

Figure 3.9 (a) shows the behaviour of our estimator for Case 1. Here we keep the skew of the join columns for the lower join fixed as 2. In the join of the upper relation, the join is with the $custkey$ column of the probe relation of the lower join. We vary the skew of the columns for the upper join. The reason why there is no graph for $z = 2$ for the upper join is that the join produced no tuples. Similar plots are observed for other values of $z$ for the lower join, and for varying domain
Figure 3.9 (b) shows the behaviour of our estimator for Case 2, i.e., when the upper join involves a join between relations that are build inputs in the pipeline. In this case, the skew of the lower join is kept fixed between columns with \( z = 1 \) and we vary the skew of the columns for the upper join. In both figures we can clearly see the fast convergence of our estimators as the probe input of the lower join is read. Similar behaviour is observed when other \( z \) values are used in the lower joins as well.

Note that in these cases, the byte and dne estimators would not have seen many tuples at the upper join by the time the probe input at the lower join has been read, since only a fraction of it would have been joined. Therefore they give inaccurate estimates.

**Joins with approximate summaries**

We now evaluate the accuracy of the approximate summary structure for join cardinality estimation introduced in Section 3.3.6. We defer an analysis of the reduced overheads due to such structures to Section 3.7.2. The summary structure utilizes a combination of bloom filters and multistage filters to approximate the distribution on the join column in the inner relation. In our experiments, we set the number of hash functions to 3 and the bits per tuple of the bloom filter to 8. We evaluate our algorithms by varying the threshold above which an element is considered a heavy hitter. We parameterize the graphs shown in Figure 3.10 using a parameter \( F \) such that the threshold is set to a fraction \( 1/F \) of the inner relation size. Likewise, we set the size of the arrays used by the multistage filter to \( 2 \times F \). Thus a larger value of \( F \) implies a lower threshold for defining a heavy hitter, and larger arrays used by the filter. In our experiments, we set the zipfian skew \( Z \) to 1.

Figure 3.10(a) shows the ratio error obtained when using multistage filters to estimate the cardinality of a join between the customer tables \( C_{1,5K} \) and \( C_{1,5K}^1 \) on the nationkey attribute. Similarly, Figure 3.10(b) corresponds to a join between the customer tables \( C_{1,125K} \) and \( C_{1,125K}^1 \) on the same attribute. In both cases, the approximate algorithms provide accurate cardinality estimates even with small samples sizes. However, we note that in Figure 3.10(b), when \( F \) is set...
### 3.7. Evaluation

We also evaluate the accuracy of the two estimators for aggregates. To choose which estimator to use we observe the value of the squared coefficient of variation ($\gamma^2$) of the group-by column. Table 3.1 contains an evaluation of the GEE and MLE estimates for tables with varying number of distinct values and skew on the TPC-H customer table with scale factor 1. The # Values column contains the maximum number of distinct values possible in that table column. The actual number of values may be much less since several values get discarded when generating highly skew data. The $\gamma^2$ column shows the value of $\gamma^2$ when 10% of the input data has been seen. We choose 10% since our final choice of the estimator is made by this point. The GEE and MLE columns show the number of input rows seen by the estimators before they reach within 10% of the correct answer. The “All Seen” column shows the number of input tuples read before all grouping values have been seen. The estimator that reaches within 10% of the correct answer by observing the smallest number of tuples is better. As the table shows, GEE usually does better than MLE on high skew data and when there exists a large number of low frequency values in the input. In other cases, MLE is a better estimator. The results validate our reasoning behind the choice of two estimators.

The table clearly shows a wide gap between $\gamma^2$ values for low skew and high skew data, and we can observe a correlation between the value of $\gamma^2$ and which estimator does better. In practice, we set a limit of 10 on $\gamma^2$, and use this as our threshold $\tau$; selecting MLE when $\gamma^2 < 10$ and GEE otherwise. We use this simple thresholding cutoff for (runtime) performance reasons and it performs well in practice. Using more sophisticated statistical tests is possible (e.g., chi-square).

<table>
<thead>
<tr>
<th># Values</th>
<th>$Z$</th>
<th>$\gamma^2$</th>
<th>GEE</th>
<th>MLE</th>
<th>All Seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
<td>0.07</td>
<td>5500</td>
<td>1400</td>
<td>6200</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10.39</td>
<td>10000</td>
<td>2300</td>
<td>16400</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>59.68</td>
<td>92300</td>
<td>104000</td>
<td>149700</td>
</tr>
<tr>
<td>5000</td>
<td>0</td>
<td>0.27</td>
<td>20100</td>
<td>6900</td>
<td>48300</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>38.62</td>
<td>33300</td>
<td>16300</td>
<td>117600</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>62.09</td>
<td>103200</td>
<td>109000</td>
<td>149800</td>
</tr>
<tr>
<td>10000</td>
<td>0</td>
<td>0.30</td>
<td>32900</td>
<td>13300</td>
<td>92200</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>50.96</td>
<td>24400</td>
<td>38300</td>
<td>149300</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>63.28</td>
<td>108200</td>
<td>117200</td>
<td>149800</td>
</tr>
<tr>
<td>25000</td>
<td>0</td>
<td>0.20</td>
<td>40600</td>
<td>33700</td>
<td>149400</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>60.85</td>
<td>63300</td>
<td>80400</td>
<td>150000</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>63.28</td>
<td>109100</td>
<td>133500</td>
<td>149600</td>
</tr>
<tr>
<td>50000</td>
<td>0</td>
<td>0.12</td>
<td>47000</td>
<td>76800</td>
<td>150000</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>64.92</td>
<td>90100</td>
<td>99700</td>
<td>150000</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>63.29</td>
<td>105600</td>
<td>113100</td>
<td>149800</td>
</tr>
</tbody>
</table>

Table 3.1: Accuracy of aggregation estimators on a 150000 row table to 100, the accuracy remains low. This is a direct result of the threshold fraction being too high. However, even in this case, the accuracy is reasonable for the purposes of query monitoring.
but that will impose additional overheads to the runtime performance of this decision.

Table 3.1 shows when our estimators converge to approximately the correct answer. However, it is also important to note the behaviour of estimators before convergence. Figure 3.11 shows the behaviour of the estimators on a group-by query on customer table $C_{(z, 10K)}$ for varying $z$. As can be seen, for $z = 0$, the GEE estimator significantly overestimates the number of groups in the input column when the sample size is small. This is because it assumes that any low frequency value is a sample of the set of all low frequency values. In this case $\gamma^2$ is low enough, and we use the MLE estimate. For $z = 1$, the GEE estimator converges more quickly and we select it because of the high $\gamma^2$ value detected in the data. In the final graph, for $z = 2$, we can see that both the estimators converge to the correct cardinality estimate almost together, after seeing a significantly large fraction of tuples. However, even in such cases, the GEE estimator is still always closer to the correct answer than the MLE estimator, which justifies our choice.

Our aggregation estimation framework gives us perfect cardinality estimates by the end of the sorting/hashing phase that precedes the aggregation. In contrast, the byte and dne estimators treat the sort/hash operator as a driver node, and thus do not start refining estimates until the sort or hash is over.
3.7. Evaluation

<table>
<thead>
<tr>
<th># Values</th>
<th>Histograms</th>
<th>Approximate summaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>205.1 KB</td>
<td>34KB</td>
</tr>
<tr>
<td>100000</td>
<td>2.03 MB</td>
<td>124KB</td>
</tr>
<tr>
<td>1000000</td>
<td>20.3 MB</td>
<td>1 MB</td>
</tr>
</tbody>
</table>

Table 3.2: Memory Overheads

3.7.2 Runtime Estimation Overheads

We now quantify the overheads imposed by our framework on the query engine. Our estimators introduce overheads in the form of extra memory consumption for maintaining histograms, and the cost of executing the extra code for probing the histograms and refining cardinality estimates. Our discussion here is in terms of the `lineitem` and `orders` tables of the TPC-H schema, since these are the largest tables as per the specification. Notice that overheads are a function of table sizes and not the table distribution. As a result, we only vary the scaling factor of the TPC-H database generated, without modifying the data distributions. A TPC-H database with a scaling factor of 1 has a 6 million row `lineitem` table and a 1.5 million row `order` table. A database with scaling factor 2 doubles these parameters, and one with scaling factor 0.5 is approximately half the size.

We first quantify the memory overheads, which are primarily in the form of histograms. We then present results detailing the runtime performance overheads to queries due to the estimators added. We used precomputed random samples in the experiments we present. We also did experiments with samples computed on the fly, with the results being consistent modulo the overhead of computing the sample. We explicitly quantify the overheads associated with dynamic runtime sampling in Section 3.7.2.

Memory Overheads

We now describe the memory overheads of our histogram implementation and compare it with the approximate summary structures which we introduce. We evaluate the structures on tables of increasing size parameterized on the number of distinct values in the column. We note that the size of the histogram depends on the number of distinct values, while the size of the summary structure depends on the size of the table. Therefore, we assume the column is a primary key column in order to provide a fair comparison between the techniques.

We implement histograms using the generic hash table structure of PostgreSQL. As Table 3.2 shows, this corresponds to a space overhead of approximately 20 bytes per entry in the histogram. However, we store just 8 bytes per entry in the histogram; 4 bytes for the value, and 4 for its count. The extra overhead is due to the pointers maintained by PostgreSQL. A simpler hash table would reduce memory costs significantly. The same table also shows the memory overheads of our approximate summary structures parameterized with $F = 1000$, 3 Hash functions, and 8 bits per tuple. The primary overhead is due to the bloom filter which has size $(tableSize \times bitsPerTuple)$. 
Chapter 3. ConEx: Query Monitoring

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Hash Joins</th>
<th>Merge Joins</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig.</td>
<td>5% Samp.</td>
</tr>
<tr>
<td>0.25</td>
<td>4m. 5s.</td>
<td>4m. 11s.</td>
</tr>
<tr>
<td>0.5</td>
<td>8m. 48s.</td>
<td>9m. 7s.</td>
</tr>
<tr>
<td>1</td>
<td>20m. 12s.</td>
<td>21m. 27s.</td>
</tr>
<tr>
<td>2</td>
<td>51m. 48s.</td>
<td>54m. 7s.</td>
</tr>
</tbody>
</table>

Table 3.3: Overheads in Hash and Merge joins

<table>
<thead>
<tr>
<th>Outer Selectivity</th>
<th>Orig.</th>
<th>5% Samp.</th>
<th>10% Samp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5%</td>
<td>2m. 30s.</td>
<td>2m. 31s.</td>
<td>2m. 33s.</td>
</tr>
<tr>
<td>1%</td>
<td>4m. 10s.</td>
<td>4m. 12s.</td>
<td>4m. 13s.</td>
</tr>
<tr>
<td>2%</td>
<td>6m. 41s.</td>
<td>6m. 43s.</td>
<td>6m 45s.</td>
</tr>
<tr>
<td>5%</td>
<td>12m. 15s.</td>
<td>12m. 22s.</td>
<td>12m. 26s.</td>
</tr>
</tbody>
</table>

Table 3.4: Overheads in Index Nested Loops joins

However, we can see that the overheads are still significantly lower than with regular histograms. In both cases, the overheads are acceptable for modern computer systems which have large memories, and do not slow down the query, as shown in the next section.

Joins

In order to evaluate the overheads of our estimators on simple binary hash and sort-merge joins, we generated lineitem and order relations from the TPC-H database with different scaling factors. We measured the time to join corresponding lineitem and order tables on the orderkey attribute. This is a primary-key foreign-key join. Thus, for example, a join between these tables with scaling factor 1, should produce a histogram of 1.5 million entries, consuming approximately 30 MB of memory. Table 3.3 shows the performance overhead experienced in both binary hash joins and sort-merge joins due to our estimators with varying sample sizes. These numbers are averages of multiple runs of the queries. As can be seen, the performance overhead of the framework is small and a small fraction of the overall query response time. This is primarily due to the fact that estimation takes place in the preprocessing phases and not during actual join processing. These phases involve heavy I/O and the overheads due to using slightly more memory are reasonable.

Table 3.5(a) describes the performance overhead due to our estimation framework on join pipelines involving joins between copies of order relations on the orderkey column. We duplicate the orderkey column in the tables in order to ensure that this corresponds to the case of joins on different attributes. The table shows overheads when the join column for the upper join involves the probe input of the lower join (Case 1) and when it involves the build input (Case 2). The sample size was set at 10%, and the overheads are slightly lower for smaller sample sizes. In both cases, our estimation framework does not introduce any major overheads and query times are increased imperceptibly.

We also evaluate the costs of cardinality estimation for Index Nested-Loops joins. Since these
3.7. Evaluation

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Orig.</th>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>1m. 10s.</td>
<td>1m. 14s.</td>
<td>1m. 16s.</td>
</tr>
<tr>
<td>0.5</td>
<td>3m. 36s.</td>
<td>3m. 39s.</td>
<td>3m. 42s.</td>
</tr>
<tr>
<td>1</td>
<td>8m. 25s.</td>
<td>8m. 52s.</td>
<td>8m. 58s.</td>
</tr>
<tr>
<td>2</td>
<td>30m. 35s.</td>
<td>31m. 5s.</td>
<td>31m. 43s.</td>
</tr>
</tbody>
</table>

(a) Join Pipelines

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Orig.</th>
<th>GEE</th>
<th>MLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>6.2s.</td>
<td>6.6s.</td>
<td>6.8s.</td>
</tr>
<tr>
<td>0.5</td>
<td>12.9s.</td>
<td>13.9s.</td>
<td>14.5s.</td>
</tr>
<tr>
<td>1</td>
<td>26.6s.</td>
<td>28.8s.</td>
<td>29.7s.</td>
</tr>
<tr>
<td>2</td>
<td>2m. 11s.</td>
<td>2m. 19s.</td>
<td>2m. 20s.</td>
</tr>
</tbody>
</table>

(b) Aggregations

Table 3.5: Overheads of Estimation

Joins are typically used for smaller data sizes, we experiment with the Lineitem and Orders tables of a TPC-H database with scaling factor 1, and vary the selectivity of a selection predicate on the outer relation (Orders). Table 3.4 displays the results of our experiment measuring the overheads of our scheme for Index Nested-Loops joins. In this experiment, a precomputed random sample of the outer relation is stored as a prefix of the table. This prefix is read and joined with the inner index to estimate the cardinality of the join. Once the cardinality estimation step is completed, the entire relation (including the random sample prefix) is read again, and joined with the inner index to produce result tuples. Despite the additional index probes made by our technique, the overheads of our scheme are low due to the relevant index pages being loaded into the cache when the prefix is joined with the index for the first time.

Aggregations

Table 3.5(b) describes overheads for a group-by query on the custkey attribute of the orders table for varying scaling factors. The queries clearly show that neither the GEE nor the MLE estimators slow down aggregations appreciably. For the MLE estimator, we set the interval of recomputation as in Algorithm 4 by setting the lower bound (l) to 0.1% of the input size, and the upper bound (u) at 3.2% of the input; we double the recomputation interval if the difference between the old and new estimates is less than 1%. These values are chosen in order to provide fine granularity for the MLE estimator without having a high overhead and have been empirically validated for different data sets. This experiment was also conducted using 10% samples of the relations. From Table 3.5(b) it is evident that the maintenance procedures of the aggregate estimators proposed herein impose small performance overheads.
Chapter 3. ConEx: Query Monitoring

Overheads of Random Sampling:

We now quantify the overheads associated with runtime random sampling, and compare it to the costs of utilizing precomputed random samples. In this experiment, we utilize the lineitem table from the TPCH database with scale factor 1. The time to perform a complete sequential scan of this table is 44096 ms as measured on our system. To compute the costs of dynamic random sampling, we modified the sequential scan operator of Postgresql to compute a block-level random sample of the specified relation at runtime, before performing a sequential scan of the relation. Figure 3.12 shows that the costs of block-level random sampling at runtime can be significant even for small random samples.

Instead, in our current implementation we lay out the data on the disk such that any fixed size prefix of the table is a random sample of the whole table. We measure the costs of reading the random sample prefix for cardinality estimation, and then scanning the entire relation (including the prefix) for usual query processing. Figure 3.12 shows that the execution time for the sequential scan in this case is only marginally higher than the execution time without random sampling. The first time the random sample prefix is read, the database system automatically stores it in the buffer cache. As a result, when we then read the entire relation (including the prefix) for query processing, the prefix is already present in the cache, and therefore the costs of reading it again are very low.

3.7.3 Progress Estimation

To illustrate the benefits of our online cardinality estimation techniques for progress estimation, we present an experiment that shows a query with many relations generating a bushy plan in Figure 3.13. For this complex 8-table query (TPCH Query 8), we present our results for a 1 GB TPCH database with Zipfian skew $Z = 2$. This is because the 8-Table query on a 10 GB database results
in a left-deep plan, with each join pipeline consisting of just a single join operation. By presenting results on a smaller database, we are able to highlight the advantages of our pushdown estimation scheme for long pipelines. For this query, the optimizer generates a bushy plan tree, followed by an aggregation. The main processing is done in a pipeline of 3 hash joins, the sizes of which are underestimated by the optimizer. When the pipeline begins, our estimation framework pushes down estimation to get accurate cardinality estimates for all the joins in the pipeline. Due to this quick adjustment, it gives correct progress estimates during the rest of the query. This illustrates the utility of our push-down estimation procedures, which perform cardinality estimation in the preprocessing phases of the join. For the sake of comparison, we also include the behaviour of the dne estimator, which does not adjust the cardinality estimate for the joins higher up in the pipeline until much later, and so it overestimates the progress for a long time. The behaviour of the byte estimator is similar and hence not shown.

### 3.8 Interface Design

In this section, we present details about the design of our interface for query monitoring. Typically, the visual interface advocated for the task of query monitoring is a progress bar based interface. Such an interface provides information about the percentage of work completed and estimated work remaining. Although such a simple interface works well for tasks like file downloads, we argue that it is too simplistic to provide informative feedback about the progress of execution of a query. This is because queries are often executed in pipelined stages separated by blocking points. A progress bar is a sufficient interface if the entire process is pipelined (such as network file download). However, if the process is broken into stages, it becomes difficult to predict the amount of work to be done by future stages. In such a setting, a simple progress bar interface provides limited, and potentially incorrect feedback.
Most database systems already provide a visual utility (EXPLAIN) which provides detailed information before query execution. The EXPLAIN interface displays the structure of the query execution plan, and provides cardinality estimates of the intermediate operators in the plan. While this is useful, the initial cardinality estimates often have errors due to incomplete statistics or wrong assumptions. Therefore, we often end up with an incorrect picture of how the query execution will proceed.

Our query monitoring system merges progress indicator technology with an EXPLAIN interface in order to provide a continuous view of query execution. Along with displaying information regarding work completed and an estimate of the work remaining like any progress bar, it also updates an EXPLAIN tree with improved cardinality estimates and information about the currently executing segments of the query plan. Therefore, we offer dynamic EXPLAIN functionality, which we argue is more informative to users and administrators.

Using an EXPLAIN tree as the basis for the interface allows us to exploit a familiar interface for the purpose of query monitoring. In addition it enables viewing query monitoring information at different levels of detail. A non-technical user might be satisfied with just a progress-bar based interface which provides information about how much work has been performed by the query and estimates of work remaining. On the other hand, a system administrator might wish to discover more details regarding the query such as incorrect cardinality estimates (which point to insufficient statistics) and data distributions. We therefore implemented an interface which provides higher level information, and also supports zooming in to access pipeline and operator level information about a query.

We now describe the various components of our interface in more detail.

### 3.8.1 Progress Indicator

We incorporate a simple progress bar which displays the percentage of work completed, and the estimated work remaining for the query. We adopt the \texttt{getnext} model of work described previously for our interface.

### 3.8.2 Query Plan Viewer

The Plan Viewer component extends the traditional EXPLAIN tree to provide a graphical view of the query execution plan. Each node in the plan is annotated with the number of tuples processed by the node, and the current cardinality estimate. These values are continuously updated during query processing. In addition, the plan is visually partitioned into pipelines (a set of operators that execute concurrently), with each pipeline marked with a different colour. This provides a visual representation of query progress in the form of completed stages which complements the progress bar. The query plan viewer forms the main window of the interface.
3.8.3 Operator Viewer

The progress indicator, and query plan viewer components described previously provide a global view of query progress. However, one might want to drill down at the operator level to view more details about query behaviour. In order to enable this functionality, we support user interaction with the Query Plan Viewer. When one clicks on a node in the query plan tree, detailed information about that node is displayed in the side panels of the interface. This information includes:

**Operator Information:** In the top panel of the operator viewer, we display information on the number of tuples processed, estimated number of tuples to be processed, and the initial optimizer estimate for the node. In addition, we display operator specific information as well.

**Cardinality Convergence:** In a different panel, we display information about the convergence of our online cardinality estimates, and how these estimates compare to the original optimizer cardinality estimates for the node. This information provides confidence guarantees on the quality of the estimates produced by our framework. A graph of convergence is displayed using which one can view the rate of convergence.

**Histograms:** In the bottom panel, we display histograms on particular columns for blocking operators. These histograms are displayed on the join column for a hash operator, a grouping column for an aggregation operator, and on sorting columns for sort operators. We display the frequencies of the most common values in the column, and the average frequency of the remaining values.

3.8.4 Additional Features

We introduce additional features that make our system particularly useful for identifying poor query execution plans. These features enable analysis during and after query execution.

**Alerts:** Our online cardinality estimation techniques provide accurate cardinality estimates early in the execution of a pipelined segment of the query execution plan. If these cardinality estimates differ significantly from the optimizer cardinality estimates we highlight the plan red to signal incorrect optimizer estimates.

**Runtime Profiling:** We segment the progress bar into colour coded segments with each segment representing a pipelined stage of query execution. This enables us to identify how much time was spent in each segment of query execution. Such runtime profiling can be used to tune the system and optimize query execution time.

3.8.5 Sample Query

Figure 3.14 shows a screenshot of our system in action. The query being executed is a join between 4 relations. The query plan viewer shows that the execution plan generated by the query optimizer is a bushy plan tree with 3 hash joins. Query execution proceeds in 5 stages, with relations being
hashed and joins being performed before the final count is computed. Currently the query is in its second pipelined stage. The node currently being clicked on is a hash node. We can see that the associated join is a one pass join (0 batches) with 1092 hash buckets in use. The histogram shows that the data is almost uniformly distributed with only a few high frequency values. The join pipeline being currently executed (pipeline 2) is highlighted to signal that the optimizer estimates are off by a significant amount. In this case, the original cardinality estimate for the hash join is 371 tuples, while the actual number of tuples produced is more than 100K. The progress monitor informs us that approximately 34% of query execution has completed. We can also observe the relative lengths of the coloured segments of the first and second pipelines. The first pipeline took a small time to execute and the work done by it is represented as the small coloured region at the left end of the progress bar. The second pipeline (which is still executing) takes comparatively more time than the first one, and therefore represents more work done.

3.9 Conclusions and Future directions

In this chapter we presented the design of a system for monitoring and predicting query behaviour in a database system. We have introduced techniques for online cardinality estimation which guarantee accurate estimates at a low overhead during query execution. Our implementation and evaluation suggests that these estimation procedures can easily be integrated into a database system without affecting query performance significantly.
The primary technical insight to be drawn from this chapter is that it is feasible to overlay cardinality estimators on standard relational query evaluation plans. We exploit this observation for the purpose of monitoring and predicting the progress of query execution here. In the next chapter, we explore how this insight leads to a new class of adaptive query processing techniques.

One of the major future directions left unexplored here is monitoring query execution in a multiprocessor environment, such as a shared-nothing system. The primary challenge here is that certain steps which take place in order in a uniprocessor environment can be parallelized. For instance, on a uniprocessor, the build input of a hash join is first partitioned, and then the probe input is processed. In a shared-nothing environment, both inputs can be partitioned simultaneously in parallel. On these lines, we have recently initiated an exploration of issues in this area utilizing a SQL query processing framework over map-reduce jobs [109].
In the previous chapter we described a framework for continuously monitoring and predicting the properties of query execution. Our focus was on refining cardinality estimates of intermediate relations at query execution time. We stated that the resulting framework could potentially be used to enable adaptive/autonomic techniques in database systems. In this chapter, we present a specific extension of the estimation framework of ConEx with applications towards the problem of adapting query execution plans at runtime.

Despite many years of research, database query optimizers often generate sub-optimal query evaluation plans. These errors typically arise due to incorrect cardinality estimates obtained from incomplete statistics and uniformity/independence assumptions. Much research has been done to address these problems, including (to give a few examples) improvements to histograms [75], sampling [59], and new schemes for collecting and maintaining statistics [118]. However, all these techniques are fundamentally limited by the design of traditional database engines which separates the optimization and execution phases of query processing.

In order to overcome these obstacles, many adaptive query processing techniques have been proposed in recent years [40]. These techniques interleave the optimization and execution phases of query processing, enabling reoptimization of sub-optimal query plans during query execution. Typically, such systems deploy a monitoring component which collects statistics during query execution and checks certain plan switching conditions, which if satisfied, trigger reoptimization.

Adaptive query processing techniques differ in terms of the scope of reoptimization considered. *Global reoptimization* techniques [82, 98] invoke the query optimizer with new statistics collected during query execution, and generate a new evaluation plan for the entire query, which may or may
not incorporate the partial results produced by the current plan. The goal of these techniques is to generate the best possible query plan, while utilizing partial statistics obtained for the subset of relations accessed at the point of reoptimization. Since these techniques utilize partial local information to perform a global plan transformation, they run the risk of cardinality estimation errors similar to those associated with traditional query optimization. In contrast, local reoptimization techniques such as the switchable plans in Rio [11], and nested loops join reordering [87], avoid such risks by altering the current plan through smaller localized changes. The goal of these techniques is to switch to a better plan, in a robust, lightweight, and risk-free manner.

In this chapter we present the XS (Execution Simulation) framework for reordering join pipelines during query execution. XS is a local reoptimization technique which detects sub-optimality of the currently executing join pipeline, and if so, reorders its inputs to obtain a better join pipeline. We introduce the concept of simulation of join pipelines using random samples and statistical summaries, and demonstrate how simulation can be used for cardinality estimation. We derive confidence guarantees, and show that our techniques provably converge to correct cardinality estimates with increasing random sample size. We present techniques for simulating all alternative execution paths of a given join pipeline efficiently without making inter-table independence assumptions enabling us to quickly identify the optimal alternative join pipeline.

4.1 Related Work

Adaptive query processing has recently received much attention from the research community. Babu and Bizarro [10] provide an overview of this area and define a taxonomy of such systems. More recently Deshpande et al. [40] also provide a comprehensive survey of this area.

Two systems similar to XS are the join reordering framework introduced by Li et al. [87] and the proactive reoptimization system, Rio [11]. Both these systems advocate lightweight local reoptimization at runtime, avoiding optimizer invocations for the entire plan. Li et al. [87] introduce a reordering technique for pipelines of Nested Loops Joins. The system monitors operator selectivities throughout query execution and reorder operators to execute the more selective joins first. However, the technique assumes independence between the selectivity of join operators, and can therefore make estimation errors and thrash between plans. In contrast, XS supports hash join pipelines as well, and avoids inter-table independence assumptions. Additionally, XS performs sampling based cardinality estimation at well defined moments of symmetry during query execution, avoiding the overhead of continuous monitoring. Like the XS framework, the Rio system [11] also performs random sample processing at runtime to switch between a set of plans. However, the notion of switchable plans is limited to switching the inner and outer inputs of a join operator. In contrast, our technique supports arbitrary reordering of the inner inputs of a join pipeline. Moreover, XS does not require any modifications to the query optimizer to produce switchable plans as required.
by the Rio system.

The Re-Opt [82] and POP [98] frameworks support global reoptimization of the current query evaluation plan. These systems detect cardinality estimation errors by monitoring the count of tuples at materialization points in the query execution plan. If the observed count is significantly different from the estimated count, reoptimization is triggered, and the optimizer is invoked with newly collected statistics. The new plan generated may or may not use intermediate results produced by the previous plan. The XS framework complements such global reoptimization techniques with local reordering techniques that do not require full plan reoptimization. In addition, we introduce advanced monitoring techniques in the form of constructing histograms at blocking points, which could also benefit global reoptimization techniques such as POP.

The Oracle RDB system [8] runs multiple plans competitively before selecting a final query evaluation plan in order to disambiguate uncertainty in execution costs. XS also compares the performance of alternative execution plans for the pipeline through lightweight simulation techniques as opposed to actually executing competing plans.

Tuple routing techniques like the Eddies framework [9] also involve exploration and evaluation of different execution plans. This comparison is performed by a central eddy operator which routes tuples and monitors costs of different operators in the plan. The work on Eddies introduced the concept of moments of symmetry, which are defined as points in the execution of a pipeline where the order of operators may be switched without affecting the correctness of the pipeline. XS takes advantage of one particular such moment of symmetry, when the inner inputs have been preprocessed, and the outer input is to be read. In contrast to XS, Eddies are continuously adaptive, and therefore require join operators with frequent moments of symmetry. Additionally, Eddies also involve the overhead of maintaining routing information for each tuple since the query may not be executed with a fixed plan. XS in contrast performs reordering only once per pipeline, if necessary, and then keeps the order of operators fixed for the remainder of query execution.

Eddies primarily found success in the context of adaptivity in data stream management systems [95]. One of the major features of join processing in such systems is that multiple data streams drive the execution of the join. As a consequence, data stream processors use symmetric hash joins [122] as a primitive operator. Symmetric hash joins were further extended to n-ary operators as MJoins [120]. These operators are primarily suited for relations that can fit in the memory of the query processor. In contrast to these works, XS operates on relational pipelines of hash joins or nested loops joins, where there is a single driver relation, and the inner tables may not completely fit in memory.

More recently, in the context of query execution feedback, Chaudhuri et al. [34] introduced techniques to estimate cardinalities along alternate execution paths of a query. Their solutions are limited to 2-way key-foreign key joins with additional restrictions on the order in which the relations are processed. Our monitoring techniques apply to general multiway equijoins, and can
be applied in the context of extracting statistics for future query executions as well.

4.2 The XS Framework

4.2.1 Problem Statement

A *pipeline* is a set of concurrently executing operators in a query execution plan. A plan is split into multiple pipelines by blocking operators. A *join pipeline* is a set of concurrently executing join operators. If the operators are the same, the join pipeline is *homogeneous*. A pipeline of \( k \) join operators has \( k \) inner inputs and 1 outer input. The outer input of a pipeline *drives* its execution.

We define the *alternative orders* of a pipeline as a set of pipelines with the same operators and inputs which have the *same outer input*. For example, the pipelines shown in Figures 4.1(a)-(c) are all alternative orders of each other (the outer input is shown on the left). We restrict the set of alternative orders to join orders not involving Cartesian products (unless unavoidable) since such join orders are typically inefficient. Alternative orders therefore capture the set of pipelines which can be easily transformed into one another by reordering *only their inner inputs*. We represent the set of alternative orders of a pipeline as a *join lattice*. Each node in the lattice represents an intermediate point along one or more join paths. The lattice shown in Figure 4.1(d) corresponds to the alternative orders shown in Figures 4.1(a),(b) and (c). We annotate nodes with an estimate of the cardinality of the corresponding intermediate join result, and edges with an estimate of the cost of the corresponding join operation. The lattice structure is described in more detail in Section 4.2.3.

We define our problem of *join reordering* as the identification of the optimal alternative order *during execution* of a given join pipeline. The definition of *alternative orders* stated previously focuses the scope of our framework to switching the order of inners in a join pipeline. Such a focus enables utilization of a well defined *moment of symmetry* at which reoptimization may be performed. This is the point when the preprocessing of the inner inputs is complete (e.g. loading the indices or sorting/hashing the inner relations), and the first *getnext* call is performed for the outer input. Reordering inner inputs at this point has no effect on the outer input (which may be a base table or materialized result of a lower pipeline) or the result of the pipeline (which may be an input for the next pipeline to be executed). Moreover, no processing of inners needs to be repeated since we do not alter the join operator and/or the outer input. Additionally, we are able to perform cardinality estimation without making independence assumptions, and without *processing join inputs out of order*. Avoiding independence assumptions is essential for accuracy purposes, while processing inputs in a given order allows our techniques to easily apply to arbitrary join pipelines in which inner and outer inputs are produced by a different processing pipeline. Our experimental evaluation in Section 4.3 demonstrates that our model of reordering significantly speeds up query execution.
In this chapter, we focus on the problem of reordering homogeneous join pipelines. In particular, we present techniques for reordering pipelines of hash joins and index nested loops joins. In Section 4.2.6, we present a brief discussion on how these techniques could potentially be extended to other join operators, and the challenges associated with non-homogeneous join pipelines.

### 4.2.2 Overview

![Join Pipelines](image)

Figure 4.1: Join Pipelines (Outer Input on the Right) and Join Lattice

Given a query evaluation plan, our system partitions it into a set of pipelines using blocking operators as partition boundaries. For each pipeline, we use the join conditions to infer the corresponding join lattice. The goal of our simulation procedures is to obtain accurate cardinality estimates at each node of the lattice. These cardinality estimates can then be utilized by cost estimation functions to obtain cost estimates for the join operations. We annotate the corresponding edges of the join lattice with these cost estimates. Given such an annotated join lattice, the problem of identifying the optimal alternative execution order is equivalent to computing the shortest
path from the bottom to the top of the lattice. In this section, we present a high level overview of our techniques, with particular emphasis on the statistical estimation procedures. Our technique proceeds in 3 steps:

**[Step 1] Obtaining a random sample of the outer input:** Our framework relies on random samples in order to obtain fast and accurate cardinality estimates. Random samples may either be precomputed (for base tables), or generated on the fly at runtime. At blocking points, we perform reservoir sampling [121] to generate samples of a certain size. Unlike the Rio framework [11], we do not require modifications to join operators to generate random samples of their outputs.

**[Step 2] All-paths simulation:** Recall the estimation procedure defined for ConEx in Section 3.3 which builds upon the sampling estimator described in Section 2.2.2. We join two relations $R \bowtie S$, and have access to a stream of tuples $r_1, r_2, \ldots, r_n, \ldots$ from $R$ generated by uniform random sampling without replacement. Let $D_n$ denote our estimate of the size of $R \bowtie S$ when $n$ tuples of a random sample of $R$ have been joined with $S$. We express $D_{n+1}$ in terms of $D_n$. If the $(n + 1)$th tuple has the value $v$ on its join attribute, it produces $N^S_v$ tuples in the join output. Therefore (ignoring confidence bounds) we have:

$$D_{n+1} = \frac{D_n \cdot n + N^S_v |R|}{n + 1}$$

We use this formula to obtain incremental cardinality estimates and associated confidence bounds at all nodes in the join lattice. Relation $R$ corresponds to the outer node of the join pipeline. Relation $S$ corresponds to a join of the inner inputs at that node in the lattice. Given a tuple $t$ of the outer relation, the primary technical challenge is to obtain the number of tuples produced $N^S_v$ for each node of the lattice. We describe our techniques to do so for general join lattices in more detail in Section 4.2.3.

Given this estimation procedure, we define stopping conditions that specify when the simulation process is to be halted. To define a stopping condition, we could utilize confidence bounds derived using limit theorems for join cardinality estimation, which however can be rather conservative. Instead, we define stopping conditions based on the stability of the estimates. At regular intervals, we call optimizer cost estimation functions with the new cardinality estimates obtained through simulation. If the resulting cost estimates remain stable within a pre-specified threshold $(1 \pm \delta)$, we conclude that they have converged to the correct values. This defines a stopping condition which enables early termination of our simulation procedure. We also bound the size of the random sample processed to ensure that the time spent in simulation is low. If the estimates haven’t stabilized during the processing of the sample, simulation is halted, and the pipeline proceeds as before without checking whether we should switch to an alternative execution plan.

**[Step 3] Comparing alternative orders and reordering:** The stopping conditions as defined previously compute the costs of joins at regular intervals and update the edges of the join lattice. In the face of these cost updates, we show in Section 4.2.7, that it is easy to dynamically maintain the
cheapest execution path on the join lattice. Once our stopping condition is satisfied, we compare the optimal alternative execution order having cost $C(Alt)$ to the current join order having cost $C(Cur)$. These costs are obtained by invoking cost functions with the new cardinality estimates. We switch to the alternative order if $C(Alt) < \tau \times C(Cur)$ where the parameter $\tau$ is a tunable threshold between 0 and 1 which represents how sensitive the reordering strategy is to the newly obtained cost estimates. This parameter is required since even with correct cardinality estimates, the cost estimation function of an optimizer is not a perfect predictor of the execution time of an operator. Once the switching decision is made, execution of the (possibly reordered) pipeline proceeds without simulating alternative orders.

Effectively, the XS framework pauses regular query execution at a moment of symmetry to explore alternative execution orders using simulation. Having simulated the alternative orders, XS selects the optimal alternative execution order, and reorders the join pipeline, if required.

**Estimation in XS vs. ConEx**

As described above, both XS and ConEx take a similar approach towards the problem of cardinality estimation. In both the systems, we exploit the structure of the query plan to perform estimation at the point when the outer input of a pipeline is to be processed. In each case, we utilize random samples to perform estimation in order to provide probabilistic confidence intervals.

The difference between XS and ConEx is in the scope of the estimation considered. ConEx is designed for the purpose of cardinality estimation only along a pipeline. For example, in Figure 4.1 (a), ConEx provides estimates for $A \bowtie B$, $A \bowtie B \bowtie C$, and $A \bowtie B \bowtie C \bowtie D$. In contrast, XS requires estimates for all possible alternative orders of a pipeline. For the same query, XS additionally requires the estimates of $A \bowtie D$ and $A \bowtie B \bowtie D$. The challenge here is to efficiently perform cardinality estimation along all alternative paths. In the next section, we describe our procedure for this.

**4.2.3 Join Simulation**

In Section 4.2.2 we described our cardinality estimation procedure [Step 2] for a 2 table join. The primary challenge we face is to obtain the number of tuples $N^S_v$ of the inner relation $S$ that join with a single tuple of the outer relation $R$ with value $v$ on the join column. For multiway joins with more than 2 relations, we require such estimates for all the nodes of the join lattice. In this section, we formally describe our simulation procedure, leaving details of specific instantiations of the procedure for different types of joins to Sections 4.2.4 and 4.2.5. We use as a running example the 3 join lattices shown in Figure 4.2. We assume that relation $A$ is the outer relation and the join initially proceeds as per the pipeline $(((A \bowtie B) \bowtie C) \bowtie D)$. The 3 join lattices represent the cases when the outer relation joins with 3, 2, and 1 inner relations respectively.
4.2. The XS Framework

We have access to a stream of random samples of the outer relation $A$. For each tuple $t(A)$ processed, we require the number of tuples it will produce at each node of the lattice. This quantity, which corresponds to $N^S_v$ is termed as the tuple-contribution $N_t(X)$ of the tuple $t$ at the node $X$. For instance $N_t(ABC) = |t(A) \bowtie (B \bowtie C)|$ is the number of tuples produced at node $ABC$ by joining tuple $t(A)$ with the inner relations $B$ and $C$. The notion of tuple-contributions also extends to tuples produced as the result of a join operation. Thus, for example, if a tuple $t(AB) \in \{t(A) \bowtie B\}$
joins with relation \( C \) to produce 3 tuples, then we write that \( N_{tB}(ABC) = 3 \). We use \( N_{tX} \) to denote tuple-contributions of a tuple \( t(AX) \in (t(A) \bowtie X) \) where \( X \) is a relation or join of relations. Our goal is to compute tuple-contributions at each node of the lattice for each tuple \( t(A) \) of the random sample of \( A \) that we process.

To compute the tuple-contributions at all nodes of the lattice, we utilize information about the structure of the lattice. Each edge of a lattice represents a join operation. We annotate edges of the lattice with \((\text{source}, \text{target})\) pairs. The target relation represents the new relation being joined at the edge. The source relation represents the relation at the lower node of the edge from which the join attribute for the edge is extracted. Thus, for example, in Figure 4.2(b), the edge between \( AD \) and \( ABD \) is annotated with \((A, B)\) since it corresponds to the join condition \( A.x = B.x \). In case of multiple choices of the source relation, we select the source relation that is lower in the join tree (the outer relation being the lowest).

Multiple edges of a lattice can have the same annotations. We term edges with the same annotations as being equivalent. Thus in figure 4.2(b), the edges \( A - AB \) and \( AD - ABD \) are equivalent. Equivalent edges capture dependencies between the tuple-contributions at each node of a lattice. For example, in Figure 4.2 (b) suppose that the current outer tuple \( t(A) \) joins with 3 tuples of relation \( B \) i.e \( N_t(AB) = 3 \). Then, we know that each tuple \( t(AD) \in \{t(A) \bowtie D\} \) would join with 3 tuples of relation \( B \) as well. If we have single equijoin conditions at each operator of the join pipeline, then a lattice representing the alternate orders of a \( k \) operator join pipeline can have its edges split into at most \( k \) equivalence classes due to the \( k \) join conditions.

Equivalence classes represent relationships between tuple-contributions at different nodes of a lattice. We maintain such relationships as a Dependency Table, which expresses the relationships between nodes in terms of sums and products. For example, consider Lattice1 in Figure 4.2(a). In this lattice, the outer relation \( A \) joins with all 3 inner relations. Therefore, we only need to compute \( N_t(AB), N_t(AC) \) and \( N_t(AD) \). The tuple-contributions at all other nodes follow from these values. For instance, \( N_t(ABC) = N_t(AB) \times N_t(AC) \), because the edges between \( A \) and \( ABC \) belong to different equivalence classes \((A, B)\) and \((A, C)\) which have the same source node \( A \). Note that this expression does not assume independence, since it represents the tuple contribution of a single tuple. Thus, if a tuple \( t(A) \) joins with 2 tuples of relation \( B \) and 3 tuples from relation \( C \), it will produce 6 tuples at node \( ABC \).

Lattice1 illustrates the case when dependencies can be expressed as only products. We now present Lattice2 from Figure 4.2(b) where sums are required as well. In this example, \( t(A) \) joins with relations \( B \) and \( D \) giving values of \( N_t(AB) \) and \( N_t(AD) \). However, the edge between relations \( B \) and \( C \) is annotated as \((B, C)\) corresponding to the join condition \( B.y = C.y \). Therefore, for each \( t(AB) \in (t(A) \bowtie B) \), we compute the number of tuples of relation \( C \) that join with it. This is its tuple-contribution \( N_{tB}(ABC) \). Summing over the values of \( N_{tB}(ABC) \) for all tuples \( t(AB) \in (t(A) \bowtie B) \) produced by the current outer tuple \( t(A) \) provides us with the value of
4.2. The XS Framework

\(N_t(ABC)\).

\textit{Lattice3} from Figure 4.2(c) represents the extreme case when the outer relation \(A\) joins with only one inner relation \(B\). The resulting tuples \(t(AB)\) can then be used to compute \(N_{tB}(ABC)\), \(N_{tB}(ABD)\) and \(N_{tB}(ABCD) = N_{tB}(ABC) \times N_{tB}(ABD)\). Summing over these values provides us with the tuple contributions \(N_t\) due to the outer tuple.

Effectively, the dependency table expresses relationships between tuple-contributions in terms of sums and products. Product terms are a result of multiple edges having the \textit{same source relation}. Sum terms are due to summation over tuple contributions produced by tuples from intermediate join results. Computing tuple-contributions correctly is at the core of our cardinality estimation procedures. We next describe how to perform such computation for index nested loops (Section 4.2.4) and hash (Section 4.2.5) joins.

4.2.4 Index Nested Loops Joins

In this section, we assume that the join lattices in Figure 4.2 represent pipelines of index nested loops joins (INLJs). We let the original join pipeline be \(((A \bowtie B) \bowtie C) \bowtie D)\), with indices on the join attribute for the inner relations \(B, C\) and \(D\), and a random sample stream of tuples from the outer relation \(A\).

Our goal is to efficiently compute tuple contributions \(N_t(X)\) at each node \(X\) of the join lattice given a tuple \(t(A)\) of the outer relation \(A\). Note that computing tuple-contributions at nodes \(A, AB, ABC\) and \(ABCD\) is straightforward. We simply execute the given join pipeline and count the number of tuples produced at each intermediate node. In contrast, computing tuple-contributions at the remaining nodes (e.g. \(AD, ABD\)) is not easy since these nodes are not observed during normal execution of the join pipeline. We can compute tuple-contributions at nodes that are not observed during normal execution of the pipeline by performing additional index probes. For example, in \textit{Lattice1} shown in Figure 4.2(a), \(N_t(AD)\) can be computed by probing the index on \(D\) with \(t(A)\).

In order to minimize the number of index probes made, we utilize the \textit{dependency table} computed for the join lattice. Consider the dependency table for \textit{Lattice1} shown in Figure 4.2(a). Given a tuple \(t(A)\) of the outer relation, we observe that there are 3 entries in the dependency table that correspond to the join of \(t(A)\) with another relation. For example \(N_t(AC) = |t(A) \bowtie C|\). Therefore, we perform index probes on \(B, C\) and \(D\) to compute \(N_t(AB), N_t(AC)\) and \(N_t(AD)\). The remaining tuple-contributions can be computed from these values.

As another example, consider the dependency table for \textit{Lattice2} shown in Figure 4.2(b). Unlike \textit{Lattice1} the outer relation in \textit{Lattice2} does not join with all inner relations. In this case, we perform index probes on relations \(B\) and \(D\) with the outer tuple \(t(A)\) to compute \(N_t(AB)\) and \(N_t(AD)\). Additionally, the tuples \(t(AB) \in (t(A) \bowtie B)\) probe the index on relation \(C\) to compute \(N_{tB}(ABC)\). The remaining tuple-contributions can be computed by appropriately summing and multiplying.
these values as per the dependency table. Similar computations can be performed for Lattice3 as well.

These examples illustrate our estimation procedure for pipelines of INLJs. We perform index-probes for each entry of the dependency table that takes the form $t \bowtie R$ where $t$ is an outer or intermediate tuple, and $R$ is an inner relation. The equations encoded in the dependency table enable us to compute the remaining values of tuple-contributions appropriately. Performing such estimation allows us to obtain tuple-contributions along all alternative execution orders without actually executing all of them.

### 4.2.5 Hash Joins

In the previous section, we described an estimation procedure for pipelines of INLJs that hinged on the fact that we have access to tuples of the inner relation through indices. If we had a pipeline of hash joins on small inputs (the entire inner hash table fits into memory), then we could have followed a similar strategy for estimation. After building in-memory hashtables for each of the inner relations, we could utilize these hashtables as indices and proceed as with INLJs. However, on larger inputs, hash join algorithms like Grace Join or Hybrid Hash Join, partition the inputs into batches and perform the join in a batchwise manner. With such algorithms, the hash join pipeline exhibits limited buffering since an outer input could either be immediately joined or written to a temporary buffer depending on the corresponding hash partition. In such cases, we cannot apply the estimation procedure in its current form since only a part of the inner hash table is accessible in memory at any point during the execution of the join.

We adapt the hash join pipeline estimation procedures for ConEx (Section 3.3.5) to the problem of all-paths estimation in XS. Our estimation procedure for a pipeline of hash join operators utilizes the fact that although inner hash tables are usually too large to fit in memory, it is possible to build a statistical summary structure on the join column which can be kept in memory for estimation purposes. We note that this summary structure is kept separately in memory from the hashtable which may be partitioned and spill to disk. The construction of this summary structure proceeds along with the hashtable creation step for the inner relations. Given the appropriate summary structures at each node of the join lattice, we can simulate the join using a random sample of the outer input as detailed in Section 4.2.2. In the discussion that follows, the terms histogram and summary structure are used interchangeably for ease of presentation. We defer a discussion on the appropriate choice of summary structures to Section 4.2.5. As before, we use as a running example the join lattices described in Figure 4.2 and assume that the initial join order is $(((A \bowtie B) \bowtie C) \bowtie D)$.

Given a join pipeline, our goal is to identify the set of statistical summary structures to be constructed for simulation of all alternative execution paths. An important consideration to note is that the original join pipeline decides the order in which relations are read. For example, given the
4.2. The XS Framework

original join order as \(((A \bowtie B) \bowtie C) \bowtie D\), the pipeline will proceed by first building hashtables on relations \(D, C\) and \(B\) in order. If these relations are larger than memory size, only some partitions are kept in memory, while the rest is spilled back to disk. This is the build phase of the hash join pipeline.

Consider \textit{Lattice1} from Figure 4.2(a). Suppose that while building inputs \(D, C\), and \(B\) we construct histograms on \(D.z\) and \(C.y\) and \(B.x\) Now, for each tuple \(t(A)\) of the outer relation \(A\), we can utilize these histograms for estimating \(N_t(AB) = |t(A) \bowtie B|\) and similarly \(N_t(AC)\) and \(N_t(AD)\). These tuple-contributions can then be used to compute tuple-contributions at the remaining nodes of the lattice.

As a somewhat different example, consider the join lattice \textit{Lattice2} in Figure 4.2(b). As before, while building inputs \(D\) and \(C\) we construct histograms on join columns \(D.z\) and \(C.y\). Similarly, while reading \(B\), we construct a histogram on the column \(B.x\). Additionally, however, we notice that there is a join condition \(B.y = C.y\) which involves \(B\) and \textit{does not involve} the outer relation \(A\). Therefore we simulate the join of \(B\) and \(C\) using the histogram on \(C.y\) to construct a new histogram on \((B \bowtie_y C).x\). This new histogram is constructed by probing the existing histogram on \(C.y\) with the value \(t(B).y\) for each tuple \(t(B)\) of relation \(B\) to get an estimate of \(|t(B) \bowtie_y C|\). This value is then used to update the frequency count of \(t(B).x\) in the histogram \((B \bowtie_y C).x\). Given a tuple \(t(A)\), we can probe this histogram to compute \(N_t(ABC)\). We can compute \(N_t(AD)\) and \(N_t(AB)\) using the histograms on \(D.z\) and \(B.x\) and use these values to compute tuple-contributions at the remaining nodes.

The above examples describe our summary structure computation algorithm. When we build a hashtable on a new inner relation \(R\) on join column \(x\), we also build a histogram on \(R.x\). Additionally, we check the dependency table for any entries marked with \(N_{tx}\) i.e entries that require tuples produced by a join with \(R\). For each of these entries, we generate histograms by simulating joins corresponding to these entries. For example, consider \textit{Lattice3} from Figure 4.2(c). As with \textit{Lattice2}, we first build histograms on join columns \(D.z\) and \(C.y\). While reading in relation \(B\), we construct a histogram on \(B.x\). Additionally, we observe entries of the form \(N_{tb}(ABC), N_{tb}(ABD), N_{tb}(ABCD)\). Correspondingly, we simulate joins with \(D.z\) and \(C.y\) to construct the histograms \((B \bowtie C).x\), \((B \bowtie D).x\) and \((B \bowtie C \bowtie D).x\), which suffice for cardinality estimation. Given a join pipeline involving \(k + 1\) relations, our procedure requires between \(k\) (e.g. \textit{Lattice1}) and \(2^{k-1}\) (e.g. \textit{Lattice3}) histograms in memory. The actual number depends on the structure of the original join pipeline and the join conditions therein. In practice, hash join pipelines which involve repartitioning have a small number of relations, and therefore few summary structures are required in general.

To summarize, we piggyback the construction of these histograms on the build phase of the join operators. The hashtable construction procedure is similar to that detailed in Section 3.3.5 (Algorithm 1). Once the inner hash tables have been built, the probe phase of the join pipeline begins. In this step, we read in a random sample of the outer input, and simulate a join of the sample
and the histograms constructed on the inner relations. Therefore, the online estimation algorithm for the case of hash joins does not involve any actual join computation. Once the sample has been processed and the stopping conditions triggered, we obtain accurate cardinality and cost estimates. We switch to our predicted optimal join order, and then perform the actual join processing. The estimation phase thus fits in between the building of the hash tables, and the join processing which involves probing the hash tables.

In the next section, we discuss our algorithms to construct approximate summary structures for the purposes of join simulation.

**Approximate Summary structures**

The discussion in the previous section assumed exact histograms as summary structures. While this is feasible for small relations, it may not be possible to store the exact distribution for larger tables. As a result, we require a technique to approximate the frequency distribution along the join column. The technique should construct the summary structure in a single pass over the data. The summary structure must answer frequency and membership queries with high accuracy. For this purpose, we exploit the summary structure introduced for ConEx in Section 3.3.6.

One might argue that utilizing an approximate summary structure exposes our technique to the same pitfalls associated with traditional query optimization. However, our techniques construct query specific summary structures, as opposed to traditional database catalog statistics, which are designed for use by all queries. As a result, we require fewer summary structures, and therefore have a larger space budget for the structures. Additionally, we construct our summary structures on the tuples after they have passed the selection predicates on the base relations, hence removing cardinality estimation errors due to the selection predicates. Finally, we note that our algorithms construct summary structures on joins of relations without making independence assumptions, hence removing a major source of error in join cardinality estimation.

**Hashtable Partitioning**

Hash Joins typically decide the number of buckets and partitions of the inner/build hash tables using the optimizer estimates multiplied by a small fudge factor. However, optimizer estimates may possibly be very inaccurate. In our experiments we found that non-optimal partitioning often leads to many tuples per bucket and significantly slows down hash join processing.

If the build input of the hash join is a relation scan with filter conditions, then we can obtain accurate estimates of the selectivities of the filters by streaming a random sample of the relation through the filters. As in the case of join pipelines, we estimate the size of the relation \( S \) passing through the selectivity predicate \( \phi \) using a random sample. If the \((n + 1)\)th tuple has the value \( i \)
\[ D_{n+1} = \frac{D_n \cdot n + I_\phi(s_{n+1}) |S|}{n + 1} \]

where \( I_\phi(s_{n+1}) \) is 1 if the \( n+1 \)th tuple satisfies the predicate \( \phi \), and 0 otherwise. Confidence bounds, as derived in Section 2.2 apply here. Using this online estimate, we can appropriately decide the partitioning mechanism for the build input, and improve the performance of the join algorithm. Additionally, this online estimation procedure also enables us to appropriately allocate space for the bloom filters and multistage filters. We note that if the inner input is the result of a join, we utilize online cardinality estimation procedures developed for progress estimation [101] in this setting.

### 4.2.6 General Pipelines

**Other Join Algorithms**

We have described techniques to perform all-paths simulation for join pipelines utilizing either existing summary structures (INLJs) or summary structures constructed at runtime (HJs). Other join algorithms fall into one or the other of these categories. For example, in a nested loops join (without indices), the entire inner input is read for each tuple (or block) of the outer. Therefore, we can create summary structures as in a pipeline of hash joins to enable all paths estimation. In a sort-merge join, the input is either sorted in advance or sorted during query processing. In the latter case, we can build summary structures during the sorting phase to enable estimation. If the input is already sorted, there is usually an index on the join column. This index can be used to do estimation as in pipelines of INLJs.

**Multiple Join Conditions**

We have described our estimation framework using single equijoin conditions at each node. We can also proceed similarly for multiple join conditions per node. In the case of pipelines of INLJs, the simulation procedure remains essentially unchanged since we have full access to tuples. For pipelines of Hash Joins, the sole change is that we now build multidimensional summary structures at a relation instead. These are built on all the join columns of the relation at that node. In the presence of space constraints, we can construct appropriate approximate histograms using techniques as described in the literature [53]. Similar extensions to the framework can also be defined for non-equijoin conditions as well.

**Non-homogeneous pipelines**

A natural question that arises in this context is the extension of join reordering techniques to non-homogeneous pipelines consisting of different kinds of join operators. The simulation techniques defined in the previous section extend to this setting as well, utilizing indexes when present, and
constructing summary structures at runtime if required. However, join reordering in this setting also includes the related problem of operator selection, wherein we may also change the operator being used at some point in the pipeline. As a consequence, reordering no longer remains a low-cost operation since operators might be switched at runtime, and new indices and hashtables may need to be loaded into memory. A detailed treatment of this question is beyond the scope of this chapter and is left for future work.

4.2.7 Dynamic Shortest Paths

The final step (Step 3) of our join reordering procedure requires us to compute the shortest paths on the join lattice. The edges of this join lattice are regularly updated with new cost estimates obtained from the updated cardinality estimates. We now describe our algorithm, which takes advantage of the lattice structure to maintain shortest paths inexpensively.

**Algorithm 5 Lattice Shortest Paths**

\[
\text{ShortestPaths (Graph } G, \text{Vertices } U) \\
// U is the set of inconsistent vertices \\
L = \text{Number of levels of the lattice} \\
\text{for } i = 1 \text{ to } L \text{ do} \\
\quad Inc = \phi \\
\quad \text{for all } u \in \text{level}(i) \cap U \text{ do} \\
\quad \quad \text{if } rhs[u] \neq d[u] \text{ then} \\
\quad \quad \quad Inc = Inc \cup u \\
\quad \quad \text{end if} \\
\quad \text{end for} \\
\quad \text{for all } u \in Inc \text{ do} \\
\quad \quad d[u] = rhs[u] \\
\quad \quad \text{for all } v \in \text{succ}(u) \text{ do} \\
\quad \quad \quad \text{Recompute } rhs[v] \\
\quad \quad \quad \text{if } rhs[v] \neq d[v] \text{ then} \\
\quad \quad \quad \quad U = U \cup v \\
\quad \quad \quad \text{end if} \\
\quad \quad \text{end for} \\
\quad \text{end for} \\
\text{end for}
\]

Suppose we ran a shortest path algorithm on the graph, and come up with a distances \( d[u] \) for each vertex \( u \). Now some changes happen in the graph (weights decrease or increase). Define
\[ \text{rhs}[u] = \min_{v \in \text{pred}(u)} d[v] + l(v, u) \]

where \( l(v, u) \) is the (possibly updated) distance of the edge from \( v \) to \( u \). We use the set \( \text{pred}(u) \) to denote the set of vertices with edges incident on \( u \). A vertex is inconsistent if \( \text{rhs}[u] \neq d[u] \). A shortest path maintenance algorithm needs to define an order in which vertices are to be processed in order to remove inconsistencies.

Algorithm 5 describes our shortest paths maintenance algorithm. Since our graph is a lattice, we process nodes in order of their level, where the level of a node in the lattice is the number of edges separating the node from the bottom. In the example lattice from Figure 4.1, the bottom (\( A \)) is at level 0 while the top node (\( ABCD \)) is at level 3. At each level, the algorithm first identifies inconsistent nodes, and then corrects the shortest path to such nodes, propagating inconsistencies up to the next level. This ensures that shortest path computation takes linear time in the number of vertices. We recompute shortest paths at regular intervals in order to maintain the optimal alternative execution order for the current join pipeline.

### 4.2.8 Summary

To summarize, \( XS \) provides a framework for obtaining cardinality estimates along multiple alternative execution orders of a join pipeline by simulation. Our simulation techniques take the form of per-tuple cardinality estimation encapsulated in the notion of tuple-contributions. We abstractly describe our techniques in the form of operations on a join lattice, and then provide concrete instantiations for pipelines of INLJs and HJs. Our simulation procedures utilize a moment of symmetry where reordering inner inputs in a lightweight fashion is feasible. In the context of INLJs, this point is when indices have been loaded and the outer relation is about to be read. In the context of HJs, this is the point where the inner relations have been built, and processing of the outer (probe) relation is to commence. At these points, we process a random sample of the outer relation and perform all-paths simulation in order to evaluate the alternative execution orders of the join pipeline. We note that this is a lightweight step, as demonstrated in our evaluation in the next section. Following the simulation step, the inner inputs are reordered if necessary, and the join is processed as in normal query execution.

### 4.3 Evaluation

In this section, we first describe our implementation and experimental setup for evaluating the performance of the \( XS \) framework. We then describe an experimental evaluation of our techniques.
4.3.1 Implementation and Setup

We implemented the XS framework inside the Postgresql 8.0 database engine and added support for automatic pipeline detection, all paths simulation, and shortest path maintenance. We let our tables be prefixed with a small random sample of the whole table. The table scan operators return tuples from the random sample before reading in the rest of the table. The actual code for reordering was implemented by switching pointers to inner indices/hashtables and modifying join conditions appropriately. Since no tuples have actually been joined at the reordering point, no state is required to be maintained after reordering. For hash joins, we implemented both exact histograms and approximate summary structures for join simulation. Unless otherwise specified, our experiments were conducted using exact histograms.

In our experiments, we utilized two standard test databases. We generated a 2.5 GB database using the DMV data generator [98, 11]. The DMV dataset consists of 4 tables. The table sizes of Owner (O) and Car (C) are 2.5 million tuples; the size of Demographics (D) is approximately 3.6 million tuples; and the size of Accidents (A) is approximately 10.7 million tuples. The dataset was generated with the correlations flag set on which makes cardinality estimation highly challenging. We utilize this dataset for all our experiments except the ones in Section 4.3.2. For the experiments in Section 4.3.2, we generated a 10 GB TPCH database. This database was generated with Zipfian skew 1 using a publicly available tool [32]. The experimental evaluation was conducted on a lightly loaded machine running Suse Linux with 4 GB memory and 3.60GHz clock speed. All queries were run with a cold cache.

Before describing the methodology of our experiments, we introduce some notation for pipelines. We label a pipeline with the order of its inputs starting with the outermost input. Thus, the pipeline shown in Figure 4.1 (a) is labeled as ABCD, and the ones in Figures 4.1 (b) and (c) are labeled ABDC and ADBC respectively. In addition, we denote the tables Owner, Car, Demographics and Accidents by O, C, D and A respectively.

There are two primary goals of our experimental evaluation.

- To show that the XS framework can detect a better alternative execution path for the pipeline if one exists.

- To show that if the current query execution plan is optimal, the XS framework detects its optimality without imposing a significant overhead on query execution.

We demonstrate this by running XS on plans generated by the Postgresql optimizer. Our experimental evaluation shows that the framework optimizes suboptimal query plans without imposing a significant overhead on queries executing with the optimal plan.
4.3.2 Experiments

3 way INL Join pipeline

Consider the query given below:

```sql
SELECT city, COUNT(*), avg(assets)
FROM owner O, car C, demographics D
WHERE C.ownerid = O.id AND D.ownerid = O.id
AND C.make = 'BMW' AND C.model = '318' AND D.salary > value
AND O.country1 = 'Germany' AND O.country3 = 'GM'
GROUP BY city;
```

The query joins the 3 relations on the `ownerid` attribute, and has additional selectivity conditions on the various relations. We vary the selectivity of the predicate on demographics (`salary > value`) by setting appropriate values. The selectivities of the other predicates are kept fixed.

Given this query, the Postgresql optimizer generates a pipeline with `C` as the outer relation, and index probes on `O` and `D` in order. This corresponds to the pipeline `COD`. The alternative plan has the order of index probes switched and is denoted as `CDO`. The optimizer selects plan `COD` irrespective of the selectivity of the predicate on `D` because it significantly underestimates the selectivities of the predicates on `C` and `O` due to independence assumptions between the correlated predicates `make = 'BMW'` and `model = '318'` and `country3 = 'GM'`. 
The execution times for plan COD selected by the optimizer for the current query can be seen in Figure 4.3 as the line labeled Optimizer. The X axis is marked with the selectivity of the predicate on the demographics relation. All other predicates are kept fixed. The query is then executed with the XS framework enabled. We test the switching condition after a random sample of the outer relation containing at least 1% of the relation is read with an additional constraint that at least 200 tuples pass the filter condition on the outer relation.

The different graphs in Figure 4.3 marked as XS $\tau = \text{val}$ correspond to the XS framework executed with different values of the switching parameter $\tau$. For the different settings of $\tau$ between 0.75 and 1, we see that XS switches to the alternative plan CDO at different selectivities. Recall that we switch to an alternative plan Alt if $C(Alt) < \tau \times C(Cur)$ where Cur is the current plan.

With $\tau = 1$, the framework switches when the selectivity on demographics relation drops below 0.6. At this point, the estimated cost of plan COD, given the new statistics, is 307812 units of work (as defined by the optimizer cost function), while the cost of the alternative plan CDO is 307696 units. In this case, the benefit of switching is low since the current plan COD takes 80s to execute, while the alternative plan CDO takes 75s to execute. Thus, setting the threshold condition to $\tau = 1$ defines an aggressive switching policy that forces a change in the execution plan as soon as any potential benefit is detected. However, a policy like this may lead to an incorrect plan choice since optimizer cost functions are not perfect predictors of query execution times even with accurate statistics. At the other end, when we set $\tau = 0.75$, the framework switches from COD to CDO only when the selectivity on demographics drops below 0.2. At this point, the original plan COD takes 81s to execute while the alternative plan CDO takes 64s to execute. With this policy, switching takes place only when a significant reduction in query execution time is predicted with the alternative plan. However, a conservative policy like this may miss opportunities to reorder the pipeline to a better one. In practice, we take a middle path and use a policy between these extremes, setting $\tau = 0.9$ in our remaining experiments. This enables us to utilize opportunities for reordering, while acknowledging the uncertainty in optimizer cost estimates. Finally, we note that for the queries where XS did not reorder the join pipeline, the overheads of simulation are negligible. This is because our framework performs additional index probes for only a small random sample of the outer relation.

3 Way Hash Join Pipeline

Figure 4.4 describes the execution of a similar query on O, C and D using a hash join pipeline. We vary the selectivity of the predicate on D while keeping other predicates constant. Across the range of selectivities presented in the Figure 4.4, the optimizer always selects the plan ODC as the best pipelined hash join plan. This plot is marked as Optimizer in the figure.

The plot marked with XS w/o Mem corresponds to executing the query with the join reordering enabled, but adaptive hashtable partitioning disabled. For the range of selectivity above 0.2, we
can see that XS switches to the alternative plan OCD which is significantly faster than the original query plan. The maximum improvement is at the right end of the graph for selectivity 1, where a query that originally took 201 s to execute now runs in 86 s.

The plot marked as XS shows the execution times when we enable both reordering and adaptive hashtable partitioning as described in Section 4.2.5 illustrating the additional benefits of this technique. In addition to causing significant performance gains, hashtable partitioning also ensures that our cost estimates for the different alternative execution plans are accurate. As illustrated for Nested Loops joins in Section 4.3.2, the optimizer cost functions are not perfect predictors of the query execution time. For the particular case of hash joins, the cost functions used by Postgresql assume that there are around 10 tuples per bucket of the hashtable. If the hashtable is improperly partitioned, the number of tuples per bucket may be higher. In this case, the cost function estimates do not scale properly with the number of tuples per bucket, and the function becomes a poor predictor of the cost of the join. This dependence of the cost function on the partitioning scheme illustrates an important difference between cost function requirements for a traditional query optimizer, and an adaptive query processing engine. The optimizer accepts the cardinality estimates, designs a hashtable partitioning scheme based on these estimates, and then estimates the cost of the hash join given the hashtable partitioning scheme. The adaptive query processing engine needs to estimate the costs given the pre-existing partitioning scheme designed by the query optimizer using the incorrect statistics from the catalogs. Our adaptive hashtable partitioning technique allows the execution engine to break away from the constraints imposed by the optimizer, and do a better job of partitioning the data appropriately.
3 Way Joins: Multiple Queries

We now test our framework on workloads of randomly generated queries. The queries were generated according to a fixed template with random values inserted to change the selectivities of the predicates on the different relations. For INL and Hash joins, we ran 50 queries which involve a join between \( O, C \) and \( D \). Figures 4.5 (a) and 4.5 (b) show scatter plots of the execution times for the queries with and without reordering. The X-axis shows the time taken to execute the query on an unmodified Postgresql system, when executed with the plan selected by the optimizer. The Y-axis shows the time when the query is executed using the same plan, but with the XS framework enabled. The line between the axes partitions the space into queries whose performance has improved, and queries which have become slower. Figure 4.5(c) shows the cumulative execution times for the query workloads for INL and Hash Joins.

In the case of the workload for INL Joins, 30 of the 50 queries were reordered to an alternative execution plan, with a maximum reduction of 70%. Only one query was slowed down from 12 s to 16 s due to incorrect estimates from the cost function. The cumulative execution time for the workload dropped by 36%. In the case of hash joins, we measure the benefits of reordering with adaptive hashtable partitioning. 46 of the 50 queries had execution times reduced, with 38 queries having the execution order changed, and the maximum reduction in execution time being 62%.

4 way Joins

We tested our framework on 4 way join pipelines. The join condition was a join of \( O, C, \) and \( D \) on the \textit{ownerid} attribute, and of \( C \) and \( A \) on the \textit{carid} attribute. Note that a join with the \( A \) relation cannot take place until the relation \( C \) has been joined. This is a pipeline in which the outer input does not join each inner input. For both INL and Hash joins, we varied the selectivity of the predicate on \( D \), while keeping the selection conditions on other relations fixed.

Figure 4.6(a) describes the results of our experiment with a 4 way INL join pipeline. The plot labeled \textit{Optimizer} represents the execution time of the query using the best nested-loops plan.
selected by the optimizer. The optimizer selected the plan \textit{OCAD} for the range of selectivity \( \geq 0.8 \), the plan \textit{OCDA} for selectivity between 0.2 and 0.8, and the plan \textit{ODCA} for selectivity less than 0.2. These are marked in Figure 4.6(a) using different markers for different plans. Since the outer relation remains constant over the range of selectivities, these are the 3 possible alternative plans.

When the queries are executed with the XS framework, using the plan selected by the optimizer the execution times can be seen in the dashed line in the figure marked \textit{XS}. For selectivity less than 0.8, the original optimizer plan is reordered to the optimal \textit{ODCA} plan. This is because the predicate on \( D \) becomes sufficiently selective. The optimizer overestimates the selectivities of the predicates on the other relations, and therefore makes an incorrect choice. Over the range of selectivities between 0.2 and 0.5, the query execution times are reduced by almost half.

Figure 4.6(b) describes the results of our experiments with 4 way hash join pipelines. Irrespective of the selectivity of the predicate on \( D \), the optimizer generated the plan \textit{ODCA} for all the queries. When the queries were executed with reordering enabled, the XS framework reordered the queries above Selectivity = 0.2 to the \textit{OCAD} plan. This is represented by the line marked as \textit{XS} in the figure. The maximum reduction in execution time is at selectivity 0.9 where the optimizer plan takes 290 s to execute, while the reordered plan takes 97 s.

For selectivities at and below 0.2, the XS framework does not reorder the plan selected by the optimizer. At selectivity 0.2, the optimizer plan \textit{OCAD} takes 108 s to execute while it takes 132 s if the XS framework is enabled even though it is not reordered. This is a rare case of memory management through adaptive hashtable partitioning causing more harm than good. On further investigation, we found that even though the estimates we obtain at runtime for hashtable partitioning are correct, the partitioning produced is sub-optimal due to the presence of additional unused memory in the system. Even though the optimizer originally had incorrect estimates, its hashtable partitioning was better since it had an incorrect estimate of the usable memory in the system as well.
Additionally, we also experimented with our approximate summary structures in this setting. Figure 4.6(b) shows the results of using our framework with approximate summary structures. We utilized bloom filters with 8 and 12 bits per tuple, and 3 hash functions. The threshold for the multistage filters was set as 0.1% of the size of the relation. The results are identical to the reordering algorithm using exact histograms except at one point, where the selectivity is 0.3. Here the summary structures overestimate the cost of the alternative path, and therefore the algorithm does not reorder the query plan.

Multiway joins on a 10GB TPCH database

![Graph of Multiway Joins](image)

Figure 4.7: Multiway Joins (TPCH)

Finally, we present an evaluation of our techniques on a 10 GB TPCH database generated with Zipfian skew 1 on the selection columns. Figure 4.7 shows scatterplots displaying the improvements of our techniques relative to the optimizer for workloads of 10 queries containing 4 way, 5 way and 6 way join queries. The 4 way join queries were on the Orders, Lineitem, Customer and Part tables of the TPCH schema. The 5 way queries included the Partsupp table as well, and the 6 way queries additionally included the Supplier table.

Figure 4.7(a) plots the execution times with and without reordering for queries executed with an INL Join pipeline. The figure clearly shows the benefits of our technique, with all the queries showing significant improvements in execution time. The best improvement is shown by a 4 way join query that executes in 24571 s using the join pipeline selected by the query optimizer, and in 3007 s with the XS framework enabled, demonstrating a reduction in query execution time by a factor of 8.

Similarly, Figure 4.7(b) displays the execution times with and without reordering for queries executed with a pipeline of hybrid hash joins. The original hash join pipeline was selected by the query optimizer. In this experiment, we utilized our approximate summary structures described in Section 4.2.5. We set 12 bits per tuple for the bloom filter, 3 hash functions, and threshold
of 0.1% of the relation size for the multistage filters. Our experiments demonstrate the utility of our technique for pipelined hash joins, with significant improvements for all queries. The best improvement is demonstrated by a 6 way join query whose execution time is reduced by a factor of almost 5, from 5157 s to 1096 s due to the XS framework.

4.4 Conclusions and Future Work

The primary contribution of this chapter is the idea of simulation in the context of query execution. We demonstrate that it is feasible and useful to simulate the execution of a class of alternative orders of a homogeneous join pipeline. Simulation ensures that accurate cardinality estimates are obtained without making independence assumptions. We show that such simulation leads to a local reoptimization technique which is experimentally shown to significantly reduce query execution time.

There are multiple open problems that ensue from the work described in this chapter. We list a few here:

- **Non-homogeneous pipelines and Operator switching**: The idea of simulation extends to non-homogeneous pipelines consisting of different join operators. In this situation, simulation is to be performed using both precomputed structures (such as indices), and histograms constructed on the fly. The real challenge in this scenario is to change the scope of adaptivity considered in this chapter, and consider switching operators in the pipelined plan at runtime. Potentially this might result in intermediate results being thrown away; the challenge is to quantify the risk-opportunity tradeoffs associated with this.

- **Global vs. Local Reoptimization**: XS demonstrates that it is feasible to adapt a query execution plan using small low-risk localized changes. While XS doesn’t guarantee the optimal plan, it guarantees a not-worse plan with a high degree of confidence. We envisage our local reoptimization technique working in tandem with a global reoptimization framework, with a decision making component which defines the risk-opportunity tradeoffs associated with each framework. Devising such a framework is a challenge that remains open.

- **Parallel query evaluation**: Techniques like XS are suitable for reoptimizing parallel query evaluation when there are long join pipelines executing at each node. More generally however, parallel evaluation proceeds across multiple nodes, with data being transferred between nodes, and there is little pipelining. Identifying structure and symmetry for adaptivity in such plans is a crucial first step towards extending XS to these environments.
Part II

Missing Queries
Traditional relational database systems support a boolean retrieval model, in which constraints are specified on properties of individual tuples and not on the result table as a whole. As a consequence of this model, an SQL query cannot ensure a result cardinality when executed on a given database. However, there are several situations where one would like the associated queries to satisfy a cardinality constraint on the result size. Such cases often occur in Business Intelligence applications, where analysts pose queries on existing databases.

For instance, consider a marketing scenario, in which a bank seeks to extend a promotional offer to hundred thousand young, high-income, customers in Ontario. Given a customer database, one can express these conditions as predicates on age, income and province attributes. However, note that the requirements such as young and high-income are loosely defined. For example, a marketing analyst might generate the query:

Select * from Customer
Where age < 40 and income > 10000
and province = 'ON'

However, this query might return a million tuples. On a second attempt, the analyst might choose to restrict the predicates further, and choose to focus only on the city of Toronto, obtaining the query:

Select * from Customer
Chapter 5. Stretch 'n' Shrink: Interactive Query Refinement

Where age < 25 and income > 100000
and province = 'ON' and city='Toronto'

This query, in contrast may return just five thousand tuples. Repeating this process multiple times, with each step involving an expensive evaluation of the query on the customer database, the analyst finally comes up with a query:

Select * from Customer
Where age < 30 and income > 50000
and province = 'ON'

that returns hundred thousand tuples

We contend that this process of generating analytical queries by trial and error is common and expensive. Our objective is to provide a framework to guide a user towards a query that satisfy requirements on the result cardinality. Today’s relational database systems lack support for tying the query specification to its output cardinality. In this work, we seek to address this gap.

We refer to the problem of queries returning too many or too few tuples as the many/few answers problem. For the many answers case, Carey and Kossmann [23] proposed a STOP AFTER operator in SQL to limit the cardinality of a query. Typically, this clause is combined with an ORDER BY clause, resulting in a Top-k query processing problem [43]. Top-k based approaches have been proposed for the few answers problem as well [6]. A fundamental requirement of these techniques is that they require a scoring function. Defining such a scoring function is often a non-trivial task especially when defined on multiple semantically distinct attributes. For example, to consider the marketing scenario described previously, a Top-k approach would require a scoring function that combines temporal (age), monetary (salary) and categorical (city) attributes to return a single score. For example:

order by (100-age) + (salary/500) + cityscore()

where the function cityscore() is

if city='Toronto' or city='Ottawa' return 20 else return 0

Such a scoring function is often difficult to craft. A primary requirement is that it should capture user preferences. To consider the query above, a bank might prefer younger customers if
it is offering a new account, while it might prefer high-income customers if it is extending a new investment opportunity. Defining scoring functions to express such application-specific preferences over multiple attributes is challenging. Moreover, such functions provide at best indirect control over the set of tuples returned.

In this chapter, we propose the Stretch ‘n’ Shrink (SnS) framework for the many/few answers problem which enables interactive user-aided refinement of queries to a given result cardinality through transformations of the selection predicates. The transformations take the form of relaxations i.e stretching query predicates to increase the query cardinality, and contractions i.e shrinking query predicates to decrease the cardinality. This model of refinement offers the advantage of not requiring the user to define a separate scoring function. Therefore, it generates queries which can utilize traditional query processing primitives without the need of additional infrastructure for processing ranking queries efficiently [73]. Additionally, by casting the many/few answers problem as a predicate transformation problem, this framework is able to explicitly capture preferences on how the query is to be refined, as detailed in the following examples.

**Example 4.** Consider a query with the predicates \( \text{year} > 1990 \text{ AND cost} < 5000 \) returning too few answers. There are many ways in which the query could be refined to satisfy the target result size. For instance, one could change the predicate on \( \text{year} \) to \( \text{year} > 1980 \) while leaving the other predicate unchanged. Alternatively, one could change only the predicate on \( \text{cost} \) to \( \text{cost} < 7000 \). Or, both the predicates could be modified together to \( \text{year} > 1987 \text{ AND cost} < 6500 \). Each of these choices involves relaxing one or more of the predicates i.e increasing the selectivity of the predicates.

Similarly, for the case of categorical predicates:

**Example 5.** Consider a query with predicates \( \text{country} = 'USA' \text{ AND state} = 'California' \), returning too many answers. This query could be refined by adding additional predicates such as \( \text{city} = 'Santa Cruz' \text{ OR city} = 'Milpitas' \text{ or predicates city} = 'San Francisco' \text{ AND Zip} = '94112' \). These additional predicates further constrain the results returned by the query and reduce its cardinality to the target answer size.

These examples illustrate that there might be multiple queries that satisfy the result cardinality constraint. Our goal in this chapter is to provide a navigational framework that enables users to interactively refine queries as per their preferences. We make the following contributions in this work:

- We formally define the problem of query refinement for queries with range and/or equality predicates which return too many or too few answers.
- We introduce the SnS framework for Interactive Query Refinement which encompasses all the cases outlined above. SnS supports a novel model of query refinement which keeps a user “in the loop” guiding one towards a query that best satisfies the target cardinality.
• We describe the sampling and indexing procedures underlying SnS, which are designed to aid accurate and efficient query refinement. We present an implementation and evaluation of the framework in a database system, demonstrating its practical utility.

5.1 Related Work

There has been some research on modifying query predicates with the intent to relax the query and generate more answers. Chaudhuri [26] introduced a formal model for modifying SQL queries in order to increase their output cardinality. Similarly, Chu and Chen [36] also defined a formal model of query relaxation using a type abstraction hierarchy on attributes, and proposed an extension (CSQL) of SQL to specify such queries. However, both of these are primarily formal models, and the papers do not investigate issues in practically realizing such models, especially with respect to ensuring a target cardinality. More recently, Kadlag et al. [83] introduced algorithms for relaxing multiple predicates using multidimensional histograms. However, their technique does not consider join queries, nor does it incorporate user preferences.

The typical solution proposed in the literature to handle the many answers problem is to utilize scoring functions and return only the Top-k ranked results [43, 28]. The primary problem with this approach is the requirement of a scoring function which may not be readily available. In addition, Top-k query processing is typically performed over single tables; optimizing Top-k queries over joins is a challenging problem [73]. An alternative approach, used in the many answers case, is to compute the skyline of the query results [18]. However, skyline computation over joins is expensive and predicting the size of the skyline is difficult [47, 27].

There has been much research on detecting and relaxing queries with empty results. Agrawal et al. [6] introduced a technique utilizing ranking algorithms in this context. More recently Luo [90] proposed a method for detecting empty result queries using information collected from previously executed queries. This technique, which uses materialized view matching cannot be generalized to the few answers problem though. Similarly, Koudas et al. [85] introduce a technique for relaxing an empty query by computing the set of results which are closest to the original query as per skyline semantics. However, the method requires expensive skyline computation and cannot provide guarantees on the relaxed result size. In the context of text search, Fontoura et al. [44] recently introduced a model of query relaxation along multiple hierarchical taxonomies. However, their model incorporates a cost based procedure for relaxation, and does not incorporate individual user preferences on query relaxation.
5.2 Preliminaries

5.2.1 Model

Consider a conjunctive SPJ (Select-Project-Join) query Q with selection predicates. We consider selection predicates to include range \((<, \leq, >, \geq)\) and equality (=) predicates. Each predicate can be defined on a numeric or categorical domain. We consider a numeric domain to be any domain on which a range predicate is defined. In the rest of the chapter, for ease of presentation, we assume that the numeric domain is the domain of integers, although our techniques work for general numeric domains. We extend our model to include queries with disjunctive predicates in Section 5.5.

Unlike numeric domains, categorical domains permit only equality predicates. In this chapter, we consider the class of hierarchical categorical predicates. Categorical predicates in databases often implicitly express hierarchies e.g. \(\text{(Country, State, City, Street, No.)}\) or \(\text{(Genre, Artist, Song)}\). Each attribute is thus at some level of a hierarchy, with the root at the smallest level. For instance, \text{Country} is at level 1, and \text{Street} is at level 4. We define the level of a categorical hierarchical predicate to denote the maximum level at which a predicate is defined on the hierarchy. Thus \text{Country} = 'US' and \text{State} = 'FL' has a level of 2.

We assume knowledge of the hierarchies defined by a database schema. Such hierarchies can be specified by the user or the schema designer. We note that the presence of hierarchies is a feature and not a requirement of the SuS framework. In the absence of information about hierarchies, we treat each attribute as a single level hierarchy by itself.

Given query Q, and a target result cardinality \(k\), we generate a new query \(Q'\) by refining the selection predicates of Q. We term the predicate transformations that increase the cardinality of Q as relaxations, and the transformations that decrease the cardinality as contractions. We next define the rules for relaxing and contracting numeric and categorical predicates.

**Numeric Predicate Refinement**

Consider a numeric predicate \(P_i : x_i < C_i\). A relaxation of \(P_i\) is any predicate \(P_i' : x_i < C_i'\) s.t \(C_i' \geq C_i\). Effectively, we have \(P_i \subseteq P_i'\). Likewise, a contraction of \(P_i\) is any predicate \(P_i' : x_i < C_i'\) such that \(C_i' \leq C_i\) i.e \(P_i' \subseteq P_i\).

We can convert any predicate on a numeric domain to a predicate of the form \(x_i < C_i\). For instance, a predicate \(x_i > C_i\) can be transformed into \(-x_i < -C_i\). We consider range predicates of the form \(C_i' < x_i < C_i''\) as two separate predicates \(-x_i < -C_i'\) and \(x_i < C_i''\). In the rest of the chapter, for ease of exposition, we assume that the numeric predicates have been appropriately transformed into predicates of the form \(x_i < C_i\).
Hierarchical Categorical Predicate Refinement

We would like to define relaxation and contraction to generate supersets and subsets (respectively) of the original hierarchical predicate in an analogous fashion to numeric predicates. In the following, we use as a running example the predicate: \texttt{Country = 'US' and State = 'FL'}.

Hierarchical Relaxation: We consider two notions of relaxation, namely \textit{Expansion} and \textit{Roll-up}. Expansion denotes the process of \textit{disjunctively} adding additional predicates at the current level of the hierarchical predicate. Thus, we can expand the example predicate to \texttt{Country = 'US' and \text{(State = 'FL' OR State = 'CA')}}. Likewise, \textit{roll-up} is the process of removing all predicates from the current level of the hierarchy. Thus, the example predicate can be rolled up to obtain \texttt{Country = 'US'}. We note that this notion of roll-up is analogous to the roll-up operation on data cubes.

Hierarchical Contraction: Similar to the forms of relaxation discussed above, we consider two notions of contraction, namely \textit{shrinking} and \textit{drill-down}. Shrinking is the inverse operation of expansion, in which predicates are removed from a disjunction at the current level of the hierarchy. Drill-down is the inverse operation of roll-up, in which additional predicates are added conjunctively at the next level of the hierarchy. The example predicate can be drilled down to obtain \texttt{Country = 'US' and State = 'FL' and City = 'Miami'}.

5.2.2 Problem Definition

We have defined notions of relaxation and contraction for numeric and categorical domains. We refer to each numeric predicate or categorical hierarchy as a \textit{dimension} of the query.

\textbf{Example 6.} The query:

\texttt{Select * from T}

\texttt{Where weight < 120 and age < 40}

\texttt{and country = 'US' and state = 'FL'}

\textit{has 3 dimensions, one for each of the two range predicates and one for the hierarchy (Country, State, \ldots)}

A query is therefore relaxed or contracted along its dimensions. We use \textit{d} to represent the number of dimensions of a query. We now define the problem of \textit{Query Refinement} as:

\textbf{Definition 1. Query Refinement Problem:} For a given SPJ query \textit{Q} with conjunctive predicates, and a target result cardinality \textit{k} on database \textit{D}, generate a query \textit{Q'} satisfying the following conditions. (i) \textit{Q'} is generated by using either only relaxations or only contractions along the dimensions of \textit{Q}. (ii) \textit{Q'} when executed on \textit{D} returns \textit{k} tuples in its result. (iii) There is no other query \textit{Q''} satisfying conditions (i) and (ii) such that the user prefers \textit{Q''} over \textit{Q'}. 
5.2. Preliminaries

We note that the requirement of using only relaxations or only contractions (Condition (i)) is to prevent predicate refinements that cancel each other out. Additionally, it ensures that the space of possible refinements can be bounded, since otherwise one could transform any query to any other query using these transformations. Effectively, when \( Q \) is estimated to return fewer tuples than \( k \), \( Q \) is relaxed; when \( Q \) returns too many tuples, it is contracted.

We consider extensions of the refinement problem to queries with disjunctive predicates, Section 5.5.

The Need for Approximation

Condition (ii) in problem definition above requires the generation of a query that returns exactly \( k \) tuples. This may be difficult since there might be no query that exactly satisfies the target cardinality. Consider a simple selection query with only a single predicate \( x < 20 \) on column \( x \) containing 100 distinct values \((1, \ldots, 100)\), with each value having a frequency of 10. In this case, if the target cardinality is 505 tuples, the best one can do is to refine the predicate to \( x < 52 \) to obtain 510 tuples. Additionally recent results show that the problem of generating a query that satisfies an output cardinality is hard to solve exactly [21]. As a consequence, our framework enables user-aided exploration of the search space to return a query that best captures user preferences, with an acceptable (for the user) error in output cardinality.

5.2.3 Terminology

Given a query \( Q \) with \( d \) dimensions, we define boundaries within which the predicate along a given dimension can be relaxed or contracted. This is captured by the notions of maximal relaxations and contractions, as defined next.

Definition 2. [Maximal Relaxation] (numeric) Given a numeric predicate \( P_i : x_i < C_i \), its maximal relaxation is a predicate \( P_i^m : x_i < C_i^m \) such that \( i) C_i^m \geq C_i \). (ii) The refinement \( Q' \) of \( Q \) produced by refining only the predicate \( P_i \) to \( P_i^m \) is estimated to return at least \( k \) tuples. (iii) There is no predicate \( P_{i'}^m : x_{i'} < C_{i'}^m \) such that \( C_{i'}^m < C_i^m \) and \( P_{i'}^m \) satisfies conditions (i) and (ii). In the absence of a predicate satisfying all three conditions, the maximal relaxation is set to \( x_i < \infty \).

Effectively, for a numeric predicate, the notion of a maximal relaxation defines the boundary up to which one could relax the predicate to satisfy the target cardinality. If the predicate is relaxed further, other predicates must be contracted, violating the requirement of using only relaxations or only contractions.

An analogous notion of maximal contractions can similarly be defined for numeric predicates. A maximal contraction bounds how much one can contract a given numeric predicate to obtain the target cardinality. We use the term maximal transformation as a generic term for maximal
relaxations and contractions (depending on whether the query is to be relaxed or contracted), and denote its value as \(P_i^m : x_i < C_i^m\) or (where the context is clear) as just the constant \(C_i^m\).

**Definition 3.** [Maximal Relaxation] (categorical) Given a hierarchical categorical predicate \(P_i : x_{i1} = C_{i1} \land \ldots \land x_{il} = C_{il}\) having a level \(l\), its maximal relaxation is a predicate \(P_i^{m} : x_{i1} = C_{i1} \land \ldots \land x_{il}^{m} = C_{il}^{m}\) such that: (i) The new level \(l^{m} \leq l\). (ii) The refinement \(Q'\) of \(Q\) produced by refining only the predicate \(P_i\) to \(P_i^{m}\) is estimated to return at least \(k\) tuples. (iii) \(P_i^{m}\) is a roll-up of \(P_i\). (iv) There is no predicate \(P_i^{m'}\) that satisfies conditions (i) - (iii), and has \(l^{m'} \geq l^{m}\). In the absence of such a predicate, the level of the maximal relaxation is set to 0.

Similar to the case of numeric predicates, maximal relaxations for hierarchical categorical predicates bound the level upto which the predicates along the dimension can be rolled up. However, the analogous notion of maximal contraction is not defined for categorical hierarchies due to the multiple possible paths for drilling down a hierarchy.

Given these definitions of maximal relaxations and contractions, we next define two queries at the core of our refinement procedures.

**Definition 4.** [Extended Query] \(Q^e\) : Given a query \(Q\) with \(d\) dimensions, the extended query \(Q^e\) is the query which returns tuples satisfying the predicates of \(Q\) along at least \(d-1\) of the \(d\) dimensions.

**Example 7.** The extended query for the query in Example 6 is

```
Select * from T where
(age < 40 and country = 'US' and state = 'FL') OR
(weight < 120 and country = 'US' and state = 'FL') OR
(weight < 120 and age < 40)
```

Observe that \(Q^e\) is a superset of \(Q\); \(Q^e\) is also a superset of any query generated by relaxing \(Q\) along only one dimension.

**Definition 5.** [Bounding Query] \(Q^b\) : Given a query \(Q\) with \(d\) dimensions, the bounding query \(Q^b\) is a refinement of \(Q\) with the predicates along each dimension maximally relaxed.

If \(Q\) is to be contracted, \(Q^b\) is identical to \(Q\), and is therefore a superset of all queries generated by contracting the predicates of \(Q\). If \(Q\) is to be relaxed, \(Q^b\) is a superset of any query \(Q'\) satisfying the target cardinality constraint, where \(Q'\) is generated by relaxing the predicates of \(Q\). Therefore the bounding query \(Q^b\) bounds all possible solutions to the query refinement problem.

**Example 8.** If the query in Example 6 is to be relaxed, a possible bounding query is:

```
Select * from T
Where weight < 170 and age < 60
and country = 'US'
```

with the values in bold text indicating the maximal relaxations along each of the three dimensions.
The bounding query \( Q^b \) bounds the search space for refinements of \( Q \) that satisfy the target cardinality constraint. Generating \( Q^b \) requires computation of maximal relaxations along each dimension of \( Q \). The maximal relaxations can be computed by utilizing the extended query \( Q^e \), which is a superset of all queries that relax \( Q \) along only one dimension. Before we describe our procedures for performing such computations, we outline the cardinality estimation scheme underlying our framework.

### 5.2.4 Cardinality Estimation Scheme

Our query refinement framework requires fast and accurate cardinality estimates for any potential refinement \( Q' \) of the original query \( Q \). These estimates could be obtained from the cardinality estimation component of the database system. However, such estimates are often incorrect, especially for queries with multiple joins and selection predicates [77]. The accuracy of refinement directly depends on the cardinality estimation scheme deployed. Therefore, we utilize sampling based estimators for cardinality estimation. In order to avoid sampling repeatedly for each refinement considered, we deploy a *Superset Sampling Estimator (SSE)* which provides fast and accurate cardinality estimates.

Consider an SPJ query \( Q^s \), which we term as a *superset query*. Given \( Q^s \), SSE provides cardinality estimates for any query \( Q' \subseteq Q^s \) to within an error \( \epsilon |Q^s| \) with high probability. We next describe SSE if \( Q^s \) is a single relation or a join query.

#### Single Relation Queries

Suppose \( Q^s \) is a selection query on a single relation \( A \). Since \( Q^s \) is defined on a single relation \( A \), a random sample of \( Q^s \) can be obtained by sampling from the underlying relation \( A \), and applying the predicates of \( Q^s \). We wish to specify guarantees for using this random sample for estimating the cardinality of any query \( Q' \subseteq Q^s \).

A *range space* is a set system, defined by a set \( P \) and a set of subsets \( R \) (ranges) of \( P \). In our current problem setting, the set \( P \) is the superset query \( Q^s \) while the ranges are all possible queries \( Q' \subseteq Q^s \). An \( \epsilon \)-approximation \( E_P \) of a set \( P \) for a range space \( R \) has the property that for any \( R \in R \)

\[
\left| \frac{|P \cap R|}{|P|} - \frac{|E_P \cap R|}{|E_P|} \right| \leq \epsilon
\]

An \( \epsilon \)-approximation \( E \) thus guarantees selectivity estimation to within a \( 1 \pm \epsilon \) interval. The following lemma of Vapnik and Chervonenkis [119, 65] links the size of a random sample of a set, and the error guarantee obtained using the sample for approximate range counting.

**Lemma 2.** For any range space with finite VC dimension, a random sample of size \( O\left(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \right) \) is an \( \epsilon \)-approximation with probability \( 1 - \delta \).
This Lemma provides guarantees on the size of the random sample of $Q^*$ required to estimate the cardinality of any query $Q' \subseteq Q^*$ to within $\epsilon|Q^*$| with high probability. We note that this random sample needs to be computed only once, and can then be kept in memory.

Join Queries

The $SSE$ procedure for a single relation query relies on the fact that one can easily obtain random samples of a base relation. If the superset query $Q^*$ is a join query over multiple relations, then obtaining a uniform random sample of $Q^*$ is known to be difficult [31].

However, utilizing random samples of base relations for join cardinality estimation is a well known technique in database literature [61]. In this work we deploy the $t_{\text{index}}$ join cardinality estimation scheme [61] which obtains a random sample from the outer relation, and joins it with indexes on the other relations. We note however, that $SSE$ for joins can utilize any alternative sampling based join cardinality estimation scheme as well.

If a random sample of the outer relation of size $n$ tuples joins with the inner indexes to produce $n_{\text{join}}$ tuples, the cardinality of the join can be estimated as $n_{\text{join}} \times N_{\text{outer}}/n$ where $N_{\text{outer}}$ is the size of the outer relation. This is identical to the estimation schemes for ConEx (Chapter 3) and XS (Chapter 4) which have confidence bounds as described in Section 2.2.2.

The $t_{\text{index}}$ procedure described here provides a useful means to obtain accurate cardinality estimates for any join query. However, the cost of performing such estimation can be prohibitively high if a join is performed with the inner indexes for each query for which a cardinality estimate is required. Instead, $SSE$ executes the $t_{\text{index}}$ procedure only once for the superset query $Q^*$. The tuples produced by this procedure are stored in an in-memory data structure. This set of tuples serves as an $\epsilon$-approximation $E_{Q^*}$ for estimating the cardinality of any query $Q' \subseteq Q^*$ to within $\epsilon|Q^*|$ with confidence bounds described in Section 2.2.2.

Given a superset query $Q^*$, a target error bound $\epsilon$, and a confidence probability $1 - \delta$, an invocation of $SSE$ with $Q^*$ i.e $SSE(Q^*)$ generates an $\epsilon$-approximation $E_{Q^*}$ for the purposes of estimating the cardinality of all queries $Q' \subseteq Q^*$. The primary advantage of utilizing $SSE$ is that one can tune the parameters to obtain estimates of desired accuracy, and avoid making any independence assumptions. Our query refinement framework invokes either of the two versions of $SSE$ described here, depending on whether the original query is a single relation query or a join query.

5.2.5 The Stretch ‘n’ Shrink Framework

We now provide a high level overview of the Stretch ‘n’ Shrink (SnS) framework for Interactive Query Refinement. SnS refines a query in two phases, with each phase utilizing the $SSE$ procedure described in Section 5.2.4.
5.2. Preliminaries

Phase 1: Computing Bounds

The goal of the first phase is to:

- Estimate the cardinality of the original query $Q$ and identify whether it is to be relaxed or contracted.

- Compute maximal relaxations and contractions along each dimension of $Q$ and generate the bounding query $Q^b$.

In order to perform such computation, $SnS$ invokes $SSE$ with the extended query $Q^e$ as the superset query to generate an $\epsilon$-approximation $E_{Q^e}$. Since the original query $Q \subseteq Q^e$, one can estimate the cardinality of $Q$ using $E_{Q^e}$, and identify whether the query is to be relaxed or contracted. Maximal relaxations and contractions can similarly be computed using $E_{Q^e}$, since they correspond to queries which relax or contract the original query along only one dimension. $SnS$ generates the bounding query $Q^b$ by refining $Q$ to its maximal relaxations along each dimension. We provide further details of the first phase in Section 5.3. We note that this phase does not require any user intervention.

Phase 2: Query Refinement

Phase 2 of $SnS$ takes as input the bounding query $Q^b$ computed in Phase 1. $Q^b$ is guaranteed to be a superset of all possible refinements of $Q$. Therefore, $SnS$ can invoke $SSE$ with $Q^b$ as the superset query to compute an $\epsilon$-approximation $E_{Q^b}$ which is utilized to estimate the cardinality of any query $Q' \subseteq Q^b$.

As illustrated in Examples 4 and 5, there might be multiple possible refinements of the original query that satisfy the target cardinality constraint. Therefore, $SnS$ provides an interactive procedure which takes into account user feedback to refine the query as per one’s preferences. There are two components of this interactive procedure:

- **Index structures for cardinality estimation:** $SnS$ utilizes the in-memory set $E_{Q^b}$ consisting of tuples produced by $SSE(Q^b)$ to compute cardinality estimates for each possible refinement $Q'$ considered by the procedure. For efficiency purposes, we devise indexing schemes over $E_{Q^b}$ which are tailored to the needs of our refinement framework.

- **Navigation Scheme:** Our goal is to provide a means for users to interactively refine queries. Therefore, we devise user navigation schemes which enable one to explore the search space for refinements $Q'$ that best capture one’s preferences.

We provide further details of our index structures and navigation schemes for queries with only numeric, only categorical, and both numeric and categorical predicates in Section 5.4.
Algorithm 6 Binary Search for Maximal Transformations

Var $D$: Database
Var $Q$: Query
Var $k$: Target

$MaxTrans(Point \ p)$

$i = \text{UnknownDimension}(p)$. \\
$\text{min} = C^l_i$ Set min, and max \\
$\text{max} = C^u_i$ to domain boundaries

while (min ≤ max) do

val = (min+max)/2
$p[i] = \text{val}$ New value on dim $i$
$Q' = \text{GenQuery}(p)$. \\
Est = CardEst$(Q',D)$; Est. card. of resulting query

if Est < $k$ then

min = val

else if (Est > $k$) then

max = val

else

return $C^m_i = \text{val}$

end if

end while End of loop for binary search

return $C^m_i = \text{val}$

5.3 Phase 1: Computing Bounds

In Phase 1, SnS identifies whether the original query $Q$ is to be relaxed or contracted, and computes maximal relaxations or contractions along all dimensions of $Q$. For this purpose, it generates the extended query $Q^e$, and invokes $SSE(Q^e)$ generating an in-memory $\epsilon$-approximation $E_{Q^e}$. $E_{Q^e}$ can be utilized to estimate the cardinality of $Q$, since $Q \subseteq Q^e$. Additionally, $E_{Q^e}$ enables computation of the maximal relaxations and contractions of $Q$ as described next.

5.3.1 Maximal Transformations (Numeric)

Consider a numeric predicate $P_i : x_i < C_i$. The SnS framework computes its maximal transformation $P^m_i : x_i < C^m_i$ through procedure $MaxTrans$ illustrated as Algorithm 6. In order to compute $P^m_i$, the procedure requires as input the predicates along the remaining $d − 1$ dimensions of the original query, with dimension $i$ set as unknown. $MaxTrans$ performs a binary search between the
5.3. Phase 1: Computing Bounds

Figure 5.1: Maximal relaxations and contractions for numeric predicates

lower ($C^l_i$) and upper ($C^u_i$) bounds of the domain of dimension $i$. For each value $val$ considered, the procedure refines predicate $P_i$ to $x_i < val$, and invokes the cardinality estimation component (encapsulated as CardEst) to obtain a cardinality estimate for the resulting query.

MaxTrans returns a value $C^m_i$ such that $Q$ with predicate $P_i$ refined to $x_i < C^m_i$ best satisfies the target cardinality constraint. This value is then compared to the original value $C_i$ of the predicate. If $C^m_i \geq C_i$, $P^m_i$ is the maximal relaxation along dimension $i$; if $C^m_i < C_i$, the query is to be contracted, and $P^m_i$ is the maximal contraction along dimension $i$.

The CardEst procedure utilizes the $\epsilon$-approximation $E_{Q^c}$ for cardinality estimation. Consider for instance the 2 predicate query shown in Figure 5.1 with predicates $A.x < C_x \land B.y < C_y$. The extended query has predicates $A.x < C_x \lor B.y < C_y$ which divide $Q^c$ into three regions $O$, $O_x$ and $O_y$ as shown in Figure 5.1. Observe that one only requires the tuples of $E_{Q^c}$ in the region $O + O_x$ in order to compute the maximal transformation $C^m_x$ along $A.x$; likewise one can compute $C^m_y$ using the region $O + O_y$. More generally, for a $d$ dimensional query, SnS computes $C^m_i$ by considering only the tuples in $E_{Q^c}$ which satisfy the predicates along the remaining $d - 1$ dimensions. These
tuples are used to construct an exact histogram sorted along dimension $i$ in memory. The MaxTrans procedure effectively performs a binary search on this histogram as accessed through the CardEst wrapper function.

### 5.3.2 Maximal Relaxations (Categorical)

Consider a hierarchical categorical predicate $P_i : x_{i1} = C_{i1} \land \ldots \land x_{il} = C_{il}$. As with numeric predicates, one can compute the maximal relaxation along dimension $i$ by considering the tuples in $E_{Q_e}$ which satisfy the predicates along all $d - 1$ remaining dimensions. For example, given the extended query from Example 7, one only needs to consider tuples which satisfy the predicates weight < 120 and age < 40.

For each such tuple $t$, SnS computes the maximum level $l_t$ such that $t$ satisfies all predicates of $P_i$ with level $\leq l_t$. Thus, a tuple that satisfies all levels of $P_i$ has a $l_t = l$, while a tuple that fails to satisfy even $x_{i1} = C_{i1}$ has $l_t = 0$. For the hierarchical predicate considered in Example 7, $l_t = 2$ if the tuple satisfies country = 'US' and state = 'FL'; $l_t = 1$ if it satisfies only country = 'US', with $l_t = 0$ otherwise.

SnS maintains a counter $N(i)$ for each level $0 \leq i \leq l$ of the hierarchical predicate. For each tuple $t$ generated by $SSE(Q^e)$ which satisfies all remaining $d - 1$ dimensions, SnS computes $l_t$ and increments $N(i)$ for all $0 \leq i \leq l_t$. At the end of the $SSE$ procedure, these counts are scaled up as per the sampling percentage. The level of the maximal relaxation is the level $l_m$ such that $N(l_m) \geq k$ and either $l_m = l$ or $N(l_m + 1) < k$. Accordingly $P_i^m : x_{i1} = C_{i1} \land \ldots \land x_{ilm} = C_{ilm}$ is the maximal relaxation along dimension $i$. If no such predicate is identified, SnS sets $l_m = 0$ i.e the resulting bounding query $Q^b$ has no predicate on dimension $i$. For the example predicate, if $N(2) = 20K$, $N(1) = 60K$ and $N(0) = 200K$, and the target cardinality is 50K, then the level of the maximal relaxation is 1 i.e the maximal relaxation is the predicate country='US'.

Given a query $Q$ with numeric and/or categorical predicates, SnS simultaneously computes the maximal relaxations/contractions along all dimensions of $Q$ with a single invocation of $SSE(Q^e)$ using the procedures outlined above. This generates the bounding query $Q^b$ which is utilized in the second phase of our framework, as described next.

### 5.4 Phase 2: Query Refinement

Phase 1 of SnS returns a bounding query $Q^b$ which bounds all solutions to the query refinement problem. This section describes indexing structures and navigation schemes for interactively exploring the search space defined by $Q^b$ in order to generate refinements of the original query. We first describe these techniques for numeric predicates in Section 5.4.1 and for categorical predicates in Section 5.4.2 before combining the techniques in Section 5.4.3.
5.4.1 Numeric Predicates

In this section, we assume that the query has only numeric predicates of the form $x_i < C_i$. We first state certain properties of the space enclosed by $Q^b$, before describing an indexing structure which exploits these properties. We then describe the navigation scheme supported by SnS for numeric predicates, and illustrate how the index supports the scheme.

![Diagram](image)

Figure 5.2: Query Refinement: Phase 2. Every point on the curve in $O^0_2$ dominates $k$ points.

If the original query $Q$ is to be relaxed, let variables $C^b_i = C^m_i$ and $C^s_i = C_i$. If it is to be contracted, let $C^b_i = C_i$ and $C^s_i = C^m_i$. Effectively, $C^b_i$ corresponds to the predicate along dimension $i (x_i < C^b_i)$ of the bounding query $Q^b$, while $C^s_i$ corresponds to the smaller of $C_i$ and $C^m_i$.

In the following we use the terms rectangle and hyperrectangle interchangeably. The bounding query $Q^b$ corresponds to a $d$ dimensional hyperrectangle $O^b: \forall_i C^l_i < x_i < C^b_i$ where $C^l_i$ is the lower bound of the domain of $x_i$.

Consider the $d$ hyperplanes $\langle x_i = C^s_i \rangle$. Each hyperplane splits $O^b$ into 2 halves, and therefore the hyperplanes along the $d$ dimensions result in $2^d$ smaller rectangles. Out of these $2^d$ rectangles, one is of particular interest.

**Definition 6.** [Dominant rectangle] ($O^0_d$): This is the rectangle enclosing the region $\forall_i: C^s_i < x_i < C^b_i$.

**Property 1.** Every solution to the query refinement problem is defined by a set of selection predicates corresponding to a point inside the dominant rectangle $O^0_d$.

This property holds because $C^s_i$ corresponds to the maximal contraction along a dimension.
**Definition 7.** \(l\)-dominated rectangle \( (O_d^l) \) Any rectangle for which there exists \( l \) dimensions \( L = \{j_1 \ldots j_l\} \) such that for any point \( x \in O_d^l \)

\[ x \in O_d^l \rightarrow \forall i \in L, x_i < C_i^s \]

is an \( l \)-dominated rectangle.

Unlike the dominant rectangle, an \( l \)-dominated rectangle is not unique. For instance, there are 2 1-dominated rectangles in Figure 5.2.

**Property 2.** Any point in \( O_d^0 \) dominates any point in \( O_d^l \) along at least \( l \) dimensions i.e is larger on \( l \) dimensions.

We next describe our index structures, which exploit these properties to optimize space requirements.

**Index Structure**

**Algorithm 7** QuadTree Insertion

\[
\text{QTInsert}(Tree,tup) \\
\text{if (Tree.isroot)} \quad \text{N++} \\
\text{Tree.elements++} \\
\text{UpdateMinMax(Tree,tup);} \\
\text{if (Tree.isLeaf == false) then} \\
\quad \text{Child = ComputeChild(tup)} \\
\quad \text{QTInsert(Child,tup)} \\
\quad \text{if (Tree.elements < } \alpha N/2 \text{) then} \\
\quad \quad \text{Merge(Tree,Children)} \\
\text{end if} \\
\text{else} \\
\quad \text{Insert(tup)} \\
\quad \text{if (Tree.elements > } \alpha N \text{) then} \\
\quad \quad \text{Split(Tree)} \\
\text{end if} \\
\text{end if}
\]

In Phase 2, \( SnS \) invokes \( SSE(Q^b) \) to generate an \( \epsilon \)-approximation \( E_{Q^b} \). Our procedure constructs an in-memory quadtree on \( E_{Q^b} \) to support fast range counting. The quadtree structure is derived from the adaptive spatial partitioning tree introduced in [67]. Algorithm 7 describes our quadtree insertion algorithm \( QTInsert \). \( QTInsert \) maintains a target fraction \( \alpha \), such that no leaf
Algorithm 8 Index Querying Procedure

\textbf{QTQuery}(QuadTree Tree, Point p)

\begin{algorithmic}
\If{(DominatesOnAllDims(p, Tree.max))}
\State return Tree.elements
\ElsIf{(DominatesOnOneDim(Tree.min, Point p))}
\State return 0
\EndIf
\If{(Tree.isLeaf)}
\If{(CountDominatedPoints(p, Tree.elementsArray) \geq \alpha N)}
\State return CountDominatedPoints(p, Tree.elementsArray)
\ElsIf{(Tree.isLeaf)}
\State dom = 0
\ForAll{(Child \in Tree.Children)}
\State dom+=QTQuery(Child, p)
\EndFor
\EndIf
\EndIf
\State return dom
\end{algorithmic}

in the tree may contain more than an \( \alpha \) fraction of the tuples seen \((N)\). If a leaf contains more than \( \alpha N \) tuples, it is split into \( 2^d \) children. Similarly, if the leaf descendents of an intermediate node together contain less than \( \alpha N/2 \) tuples, then the subtree at the node is collapsed into a leaf node. This can happen due to incoming tuples increasing \( N \). At each node of the quadtree, \( QTInsert \) maintains the \( \text{min} \) and \( \text{max} \) values along each dimension over all the tuples in the leaves of the subtree at the node.

The space requirements of the quadtree described above can be optimized further. Property 1 asserts that all solutions to the query refinement problem must lie within the dominant rectangle \( O^0_d \). Given a \( d \) dimensional point \( p' \) corresponding to a query \( Q' \), \( SnS \) requires the quadtree index to return the number of tuples in \( E_Q^k \) that are dominated by \( p' \). Property 2 asserts that \( p' \) must dominate every point \( p' \) in \( O^l_d \) along \( l \) dimensions. Therefore, for any tuple \( t_{p'} \) represented by a point \( p' \) in \( O^l_d \), one only requires the remaining \( d - l \) dimensions to check whether \( p' \) is dominated by \( p' \) (i.e whether \( t_{p'} \in Q' \)). Since it is possible to discard the \( l \) dominated dimensions, we modify the quadtree index to exploit this property by initially splitting the root node of the tree along the \( d \) hyperplanes \((x_i = C^*_i)\). This optimization results in significant space savings. For instance, the quadtree does not keep any attributes of tuples in the \( d \)-dominated rectangle \( O^d_d \), only maintaining a counter for this node.
Navigation Scheme

We now describe the navigation scheme supported by the SnS framework. Each dimension $i$ of the original query is initially associated with a range $(C^s_i, C^b_i)$ defined by the original query predicate, and the associated maximal transformation. Our goal is to support an interactive refinement procedure which enables specification of one’s preferences on the choice of values of the refined predicates, within the constraints defined by these ranges.

The interactive refinement procedure proceeds in the form of rounds between the user and the SnS framework. Each round consists of the following two steps:

- The user selects an arbitrary predicate $P_j : x_j < C_j$ and refines it to a new predicate $P'_j : x_j < C'_j$ such that $C^s_j \leq C'_j \leq C^b_j$ i.e $C'_j$ lies within the range for dimension $j$.
- SnS recomputes the ranges $(C^s_i, C^b_i)$ for each unrefined dimension $i$ conditional on the user’s current set of predicate refinements.

Each predicate refinement further constrains the ranges of the remaining unrefined predicates. This process is repeated for $d - 1$ rounds, at which point the final unrefined predicate is fully constrained, and can be computed automatically. We illustrate this interactive refinement procedure through the following example:

**Example 9.** Consider a query with predicates $\text{year} < 1960$ and $\text{age} < 25$ and $\text{salary} < 3000$ and suppose it returns too few answers. In Phase 1, SnS computes maximal relaxations $(1980, 40, 7000)$ for $\text{year}$, $\text{age}$, $\text{salary}$ respectively. In Phase 2, one may first refine the predicate on $\text{age}$ to $\text{age} < 30$. SnS in turn recomputes the ranges of the remaining predicates as $\text{year}$: $(1960,1975)$ and $\text{salary}$:$(3000,6400)$. These ranges are smaller than the ranges specified by the initial set of maximal relaxations due to the relaxation of the predicate of $\text{age}$. Given these ranges, one may refine the predicate on $\text{year}$ to $\text{year} < 1970$. SnS computes the best choice of the final predicate, relaxing it to $\text{salary} < 4100$. The final refined query has predicates $\text{year} < 1970$ and $\text{age} < 30$ and $\text{salary} < 4100$.

Algorithm 9 describes the numeric predicate refinement component ($INumRef$) of the SnS framework. $INumRef$ initially defines a range $(C^s_i, C^b_i)$ for each predicate (as per Property 1). Each predicate refinement made by the user ($GetUserRef$), results in a recomputation of the ranges for each unrefined predicate (lines 13-17). This is performed by calling the $MaxTrans$ procedure (Algorithm 6), setting the unrefined dimension $i$ as unknown (‘?’), with all remaining dimensions $j$ set to either the original $C'_j = C_j$ (if unrefined) or the refined $C'_j$ values. When only one unrefined dimension $i$ remains (line 16), $INumRef$ automatically refines the associated predicate to its maximal transformation $C^m_i$.

The version of $MaxTrans$ utilized by $INumRef$ differs from the description in Algorithm 6 in two minor ways. First, it performs a binary search over the limited range $(C^s_i, C^b_i)$. The second
5.4. Phase 2: Query Refinement

Algorithm 9 Numeric Predicate Refinement

1: \textbf{INumRef}()
2: \textbf{for all Dimensions} \textbf{i} \textbf{do}
3: \hspace{1em} \( C^a_i = \min(C_i, C^m_i); C^b_i = \max(C_i, C^m_i) \)
4: \hspace{1em} \( C'_i = C_i \)
5: \textbf{end for}
6: \textbf{if} \ \exists i : C^m_i = \infty \ \textbf{then}
7: \hspace{1em} \textbf{RecomputeLowers}()
8: \textbf{end if}
9: \textbf{NumUnRef} = d
10: \textbf{while} \textbf{NumUnRef} > 0 \textbf{do}
11: \hspace{1em} \textbf{GetUserRef}()
12: \hspace{1em} \textbf{NumUnRef} --
13: \hspace{1em} \textbf{for all UnrefinedDimensions} \textbf{i} \textbf{do}
14: \hspace{2em} \( C^m_i = \text{MaxTrans}(C'_1, \ldots, C'_{i-1}, ?, C'_{i+1}, \ldots, C'_d) \)
15: \hspace{2em} \textbf{if} Contraction \textbf{then} \( C^a_i = C^m_i \) \textbf{else} \( C^b_i = C^m_i \)
16: \hspace{2em} \textbf{if} \textbf{NumUnRef} = 1 \textbf{then} \( C'_i = C^m_i \); \textbf{return};
17: \hspace{1em} \textbf{end for}
18: \textbf{end while}
19: \textbf{RecomputeLowers}()
20: \textbf{for all Dimensions} \textbf{i} \textbf{do}
21: \hspace{1em} \textbf{temp} = \text{MaxTrans}(C^b_1, \ldots, C^b_{i-1}, ?, C^b_{i+1}, \ldots, C^b_d)
22: \hspace{1em} \textbf{if} \textbf{temp} > C^a_i \textbf{then} \( C^a_i = \text{temp}; \)
23: \hspace{1em} \textbf{end for}

difference is that the associated cardinality estimation module \textit{CardEst} now utilizes the quadtree index on \( E_Q^b \) through the function \textit{QTQuery} outlined as Algorithm 8. Given a \( d \) dimensional point \( p \), \textit{QTQuery} computes the number of tuples in the quadtree dominated by \( p \). \textit{QTQuery} also uses the \textit{min} and \textit{max} values associated with each node of the quadtree to avoid unnecessary tree traversals.

Recomputing lower bounds for relaxations: The above discussion assumes that it suffices to recompute only the maximal transformation \( C^m_i \) in each round of the refinement process. While this holds true in general, for certain special cases of query relaxation the lower bound may be too tight to achieve the target cardinality due to highly selective predicates. This is indicated by maximal relaxations for some dimensions being set to infinity. In this case, \textit{INumRef} recomputes the lowerbounds by calling the \textit{RecomputeLowers} procedure (Alg 9 lines 19-23). \textit{RecomputeLowers}
invokes \textit{MaxTrans} for each dimension \(i\), with all remaining dimensions \(j\) set to their maximal relaxations \(C^b_j\). If the returned value exceeds the current lower bound \(C^s_i\), then the lower bound is modified.

\textbf{Error Guarantees:} Our navigation scheme enables a refinement of all but one of the predicates as per user preferences. The final predicate is however refined by \(INumRef\). Let the maximum frequency of a value in dimension \(i\) in the bounding query \(Q_b\) be \(m_i\). If \(i\) is the final dimension refined by \(SnS\), then in the worst case, the resulting refined query will have an absolute error of \(\frac{m_i}{2}\) with respect to the target cardinality. This error is separate from the errors due to cardinality estimation, and is an artifact of our flexible navigation scheme. This also suggests that it is best to leave attributes with uniform distributions and many distinct values as the final unrefined dimension, since these would be expected to have a low value of \(m_i\).

5.4.2 Hierarchical Categorical Predicates

![Navigation paths for 2 hierarchies (S,C) and (G,A)](image)

In this section, we describe the indexing structures and navigation schemes supported by \(SnS\) for refining queries with hierarchical categorical predicates only. Since these predicates are equality predicates on categorical domains, one cannot utilize quadtrees and adopt a range shrinking navigation scheme as with numeric predicates. Instead, the \(SnS\) framework deploys techniques based on materializing different navigation paths on a data cube [64].

Consider a query \(Q\) with \(d\) hierarchical categorical dimensions. \(Q\) can be relaxed by \textit{roll-up} and \textit{expansion} operations applied in an arbitrary order on the upper levels of the categorical hierarchies. Likewise, \(Q\) can be contracted by \textit{shrinking} and \textit{drill-down} operations applied in an
5.4. Phase 2: Query Refinement

arbitrary order on the lower levels of the hierarchy. We refer to any feasible sequence of such relaxation or contraction operations for refining $Q$ as a navigation path or NavPath. For example, given two two-level hierarchies $(\text{State}(S), \text{City}(C))$ and $(\text{Genre}(G), \text{Artist}(A))$, the possible NavPaths for relaxation/contraction are illustrated in Figure 5.3. In the figure, an entry along a navigation path (NavEntry) of the form $G|SC$ represents a relaxation/contraction operation on dimension $G$, with predicates on $S$ and $C$ unchanged, and no predicate on $A$.

Index Structure

Algorithm 10 Constructing CFDs

```plaintext
List NavPaths = CreateNavPaths()

HistInsert(Tuple Tup)
List HistEntries = NULL
for all NavEntry Nav ∈ NavPaths do
    childAttr = ExtractChildAttr(Nav,Tup)
    HistEntry parent = GetParent(Entries,Nav)
    if hasChild(parent,childAttr) then
        HistEntry child = GetChild(parent, childAttr);
    else
        HistEntry child = AddChild(parent,childAttr);
    end if
    child.freq++
    Entries.add(child)
end for
```

As with numeric predicates, $SnS$ invokes $SSE(Q^b)$ in order to generate an in-memory $\epsilon$-approximation $E_{Q^b}$ for the purposes of cardinality estimation. The goal of our indexing structure is to support fast cardinality estimation for the operations of relaxing or contracting a hierarchical predicate by utilizing $E_{Q^b}$. $SnS$ accomplishes this by maintaining conditional frequency distributions (CFDs) for each NavEntry along any possible NavPath for relaxation or contraction.

Algorithm 10 describes our procedure for constructing CFDs as exact histograms over $E_{Q^b}$. Given a query $Q$ with $d$ hierarchies, function CreateNavPaths constructs the possible NavPaths for query refinement. Having identified the navigation paths, $SnS$ invokes function HistInsert for each tuple $Tup$ produced by $SSE(Q^b)$. HistInsert traverses the NavPaths in a breadth-first top-down fashion. Each NavEntry encountered encodes a particular CFD; for instance $G|S$ represents the frequency distribution of Genre conditional on a given value of the State predicate. Given a NavEntry, HistInsert identifies the appropriate parent and child histogram entries (HistEntry) and
Chapter 5. Stretch ‘n’ Shrink: Interactive Query Refinement

increments the frequency of the child accordingly. For instance, given a tuple with $\text{Genre} = \text{’Pop’}$ and $\text{State} = \text{’FL’}$, for $\text{NavEntry } G | S$, $\text{HistInsert } (a)$ identifies the parent $\text{HistEntry}$ for $\text{State} = \text{’FL’}$. $(b)$ looks up its children for a $\text{HistEntry}$ corresponding to $\text{Genre} = \text{’Pop’}$, creating a new child if necessary and $(c)$ increments the frequency count of the child.

Navigation Scheme

As with numeric predicates, $\text{SnS}$ supports a navigation scheme for hierarchical categorical predicates which proceeds in the form of rounds. The process begins with all predicates as in the original query. In each round:

- The user selects an arbitrary hierarchical predicate, and either rolls-up/expands the predicate (in the case of relaxation) or drills-down/shrinks it (for contractions).

- $\text{SnS}$ identifies the current position of the user along the $\text{NavPaths}$, and displays the appropriate set of CFDs for any possible next refinement step, utilizing the index structure.

This process continues until either $(a)$ the refined query exceeds the target cardinality (for relaxations) or falls below the target cardinality (for contractions) or $(b)$ one identifies a final predicate, for which $\text{SnS}$ computes an appropriate relaxation or contraction to best satisfy the target cardinality. We illustrate this process through the following example featuring only drill-down operations:

**Example 10.** Consider a query with initially no predicates which is to be contracted to a target cardinality of $50K$ tuples along the hierarchies $(S, C)$ and $(G, S)$. $\text{SnS}$ initially presents frequency distributions on $\text{State}$ e.g. (‘FL’, 10M), (‘CA’, 7M),…, and $\text{Genre}$ e.g. (‘rock’, 5M), (‘pop’, 3M),…, One may then select $\text{State} = \text{’FL’}$. In response, $\text{SnS}$ presents frequency distributions on $\text{city}$ e.g. (‘Miami’, 800K), (‘Tampa’, 400K),… and $\text{genre}$ e.g. (‘rock’, 2M), (‘pop’, 1M),…, These distributions are conditional on the selection of $\text{State} = \text{’FL’}$. This process is repeated, with one possibly selecting $\text{genre} = \text{’Rock’}$ and $\text{artist} = \text{’Coldplay’}$. The final predicate for attribute $\text{city}$ is computed by $\text{SnS}$ as $\text{city} = \text{’Miami’ OR city} = \text{’Tampa’ OR city} = \text{’Alachua’}$. The final query is therefore:

```
Select * from Album-Sales where
Genre = ‘Rock’ and
Artist = ‘Coldplay’ and
State = ‘FL’ and
(city='Miami' OR city='Tampa' OR city='Alachua')
```

Identifying the appropriate set of CFDs to display is straightforward At each round of the refinement process, the user is at some $\text{NavEntry}$, with the current set of predicate refinements corresponding to an associated $\text{HistEntry hist}$. Therefore for roll-ups, $\text{SnS}$ needs to display frequencies.
of the parents of hist. Similarly, drill-downs require displaying frequency distributions of the children of hist. Likewise, expansion and shrinking require displaying the frequency distributions of the siblings of hist and of its children respectively.

We now describe how SnS refines the final predicate.

**Refining the final predicates:** Suppose the final predicate to be refined is on attribute \( x_i \) with possible values \( C_{i1}^1, \ldots, C_{in}^n \) with associated (through HistEntries) conditional frequency estimates \( f_{i1}^1, \ldots, f_{in}^n \) respectively. The goal of this step is to *disjunctively* select a subset \( J = j_1, \ldots, j_r \) of these \( n \) values such that \( \sum_{j \in J} f_{ij}^i \approx k \). This is the subset-sum problem, which is known to be NP-hard. Although polynomial time approximation schemes exist for this problem [72, 99], we implement a greedy 2-optimal approximation algorithm [99] described as Algorithm 11. This approximation guarantee is a worst-case guarantee, and in practice the greedy algorithm works extremely well.

Procedure \( \text{FinalPred} \) (Algorithm 11) takes as input a list of possible values \( C_{i1}^1, \ldots, C_{in}^n \) for the final predicate sorted in decreasing order of their frequency \( f_{ij}^i \). \( \text{FinalPred} \) greedily adds new values to the current set (\( \text{CurrSet} \)) of values unless the associated sum \( \text{CurrSum} \) exceeds the target cardinality. The procedure ensures that \( \text{CurrSum} \) is guaranteed to be \( \leq k \). Additionally, \( \text{FinalPred} \) maintains the greedy set \( \text{CurrBigSet} \) with minimal error which has a sum \( \text{CurrBigSum} > k \). \( \text{FinalPred} \) returns either of \( \text{CurrSet} \) or \( \text{CurrBigSet} \) having minimum error.

**Algorithm 11** Refining the Final Predicate

```
FinalPred(SortedList values)

CurrSet = NULL; CurrSum = 0; CurrBigSet = NULL; CurrBigSum = 0;
for all \((C_{ij}^i, f_{ij}^i) \in \text{values}\) do
    if CurrSum + \(f_{ij}^i\) \(\leq k\) then
        CurrSet.add(C_{ij}^i)
        CurrSum += \(f_{ij}^i\)
    else if \(|\text{CurrSum} + f_{ij}^i - k| < |\text{CurrBigSum} - k|< k\) then
        CurrBigSet = CurrSet + C_{ij}^i
        CurrBigSum = CurrSum + \(f_{ij}^i\).
    end if
end for
return minErr(CurrSet,CurrBigSet)
```

To summarize, SnS supports a navigation scheme based on roll-ups/expansions or drill-downs/shrinking along multiple navigation paths for a set of hierarchies. It supports these operations by maintaining conditional frequency distributions along all such possible navigation paths.
5.4.3 Combining the techniques

In the previous two sections, we have described index structures and navigation schemes for queries with only numeric, and only hierarchical categorical predicates. In this section, we illustrate how these techniques can be combined for queries with both numeric and categorical predicates.

Index Structure

The goal of the index structure is to efficiently support cardinality estimation for range and equality queries over multidimensional categorical and numeric data. The index is built on the $\varepsilon$-approximation $E_{Q^b}$ generated by the SSE procedure.

SnS combines the quadtree index (for numeric predicates) and navigation path index (for categorical hierarchies) by adding a quadtree to each histogram entry (HistEntry) to represent the numeric dimensions of the tuples corresponding to the HistEntry. This is accomplished by adding an extra function call $QTInsert(child.Tree,Tup)$ after line 13 of the HistInsert procedure in Algorithm 10. Thus, for instance, the numeric attributes for all tuples with state='FL' are indexed by a quadtree associated with the corresponding HistEntry. Additionally, SnS maintains a global quadtree which indexes all tuples in the bounding query. This index structure suffices to provide fast cardinality estimates for any $Q' \subseteq Q^b$.

Navigation Scheme

The navigation scheme for queries with both numeric and categorical predicates remains essentially unchanged. As before, refinement proceeds in rounds. In each round:

- The user selects either a numeric or categorical predicate and refines it to a new value.
- SnS responds by updating the ranges for unrefined numeric predicates, and the appropriate CFDs for the categorical predicates.

Suppose a numeric predicate $P_i : x_i < C_i$ is refined to a new value $C_i'$. SnS recomputes the ranges of the remaining unrefined predicates by calling the MaxTrans procedure (Algorithm 6). For the categorical predicates, the relevant CFDs are updated by calling the QTQuery procedure (Algorithm 8) for the quadtrees associated with each HistEntry. Likewise, a similar procedure is adopted when a hierarchical categorical predicate is relaxed or contracted.

5.5 Disjunctive Predicates

In this section, we demonstrate how the refinement model and techniques defined previously extend to queries with disjunctive predicates. Consider the following query, as a running example through this section:
Example 11. The query:

\[ \text{Select * from T} \]
\[ \text{Where (weight < 120 OR age < 40) } \]
\[ \text{and country = 'US' and (state = 'FL' OR state = 'CA')} \]

has 3 dimensions, one for each of the two range predicates, and one for the disjunctive predicates defined on the hierarchy (Country, State, ...).

The predicates can be modified along these 3 dimensions using the same relaxation and contraction operations as defined in Section 5.2.1. The operations remain well defined, and are not affected by the predicate being in a conjunction or disjunction with the remaining predicates. Likewise, the definitions of maximal relaxations for numeric and categorical predicates as defined in Section 5.2.3 remain valid. As before, we transform only the predicate under consideration without modifying the other predicates. The maximal relaxations can be computed using the extended query. In the case of queries with disjunctions, the extended query is defined by replacing the predicates along each dimension with \( \text{TRUE} \), and taking the union of the resulting \( d-1 \) queries.

Example 12. The extended query for the query in Example 11 is

\[ \text{Select * from T} \]
\[ \text{Where ((TRUE OR age < 40) and country = 'US' and (state = 'FL' OR state = 'CA'))} \]
\[ \text{OR ((TRUE OR weight < 120) and country = 'US' and (state = 'FL' OR state = 'CA'))} \]
\[ \text{OR ((weight < 120 OR age < 40) and TRUE)} \]

This can be rewritten as:

\[ \text{Select * from T} \]
\[ \text{Where (country = 'US' and (state = 'FL' OR state = 'CA'))} \]
\[ \text{OR weight < 120} \]
\[ \text{OR age < 40} \]

Given this extended query, the maximal relaxations transformations along each dimension can be computed as in Section 5.3. The bounding query can be generated by replacing each of the predicates in the original query by the corresponding maximal relaxations. Continuing with the example under consideration

Example 13. Suppose the query in Example 11 is to be relaxed. A possible bounding query is:

\[ \text{Select * from T} \]
\[ \text{Where (weight < 200 OR age < 50) } \]
\[ \text{and country = 'US'} \]
The bounding query is executed using a random sample of a base table to obtain an in-memory \( \epsilon \)-approximation. As before, data along the numeric dimensions is stored in a quadtree (Section 5.4.1), while the categorical data is stored using an index along the navigational paths (Section 5.4.2).

The only distinction between the schemes for conjunctive and disjunctive queries is that in the case of numeric data, a conjunctive query corresponds to a single multidimensional point, as described in Section 5.4.1, Algorithm 8. Estimating the cardinality of a query with conjunctive numeric predicates simply required counting the number of points inside a rectangle using the quadtree. In contrast, a disjunction of range predicates corresponds to a union over multiple rectangles. Such queries too can be handled easily by the quadtree data structure, by ensuring that the disjunctive query is split into disjoint rectangles, and summing up the cardinality estimates for each of the rectangles. In contrast to numeric predicates, the procedure for categorical data remains unchanged, since we index all possible navigation paths for relaxation or contraction as before.

This section demonstrates that our refinement framework can easily incorporate queries with disjunctive predicates. In the next section, we describe an evaluation of a prototype implementation of our framework in a database system.

5.6 Evaluation

In this section, we describe an implementation and experimental evaluation of our refinement framework. We implemented the \( SSE \) procedure in the Postgresql 8.0 database system, and implemented the \( SnS \) framework as a Java based frontend. \( SnS \) communicates with \( SSE \) through the JDBC layer and network sockets. Each query \( Q \) submitted for refinement results in two invocations of the \( SSE \) layer, once for the extended query \( Q^e \), and once for the bounding query \( Q^b \). Our instantiation of \( SSE \) utilizes precomputed random samples on base relations. In our evaluation, the \( SSE \) procedure is halted when it generates an in-memory subset of the result of size 5000 tuples, with a sampling threshold of at most 5\%. Varying the sample size (provided it isn’t too low) did not have a significant affect on the cardinality estimates.

We conducted experiments on three different test databases. The primary test database is a 2.5 GB database generated using the DMV data generator [98]. The DMV dataset consists of 4 tables. The table sizes of \( Owner (O) \) and \( Car (C) \) are 2.5 million tuples; the size of \( Demographics (D) \) is approximately 3.6 million tuples; and the size of \( Accidents (A) \) is approximately 10.7 million tuples. The dataset was generated with the correlations flag set on. This produces many interesting correlations between columns on different tables, which makes cardinality estimation highly challenging. Additionally, we also performed experiments on two TPC-H databases of size 1 GB each. One of them is the standard TPC-H database generated as per the benchmark specification. Since this database consists of uniformly distributed data, we also generated a TPC-H database
with zipfian skew $Z = 1$ using a publicly available tool [32].

To simulate user interaction with the system, we implemented an external function which randomly refines predicates (within the constraints imposed by the framework) at each round of the refinement process. In our experiments we plot the relative error of the refined queries, which is defined as

$$Err = \frac{|\text{RefinedQueryCard} - \text{TargetCard}|}{\text{TargetCard}}$$

The experimental evaluation was conducted on a lightly loaded machine running Suse Linux with 4 GB memory and 3.60GHz clock speed.

### 5.6.1 Accuracy Experiments

![Figure 5.4: Single Query Varying Target](image)

Figure 5.4 describes the results of an experiment using the DMV database in which we fixed the original query, and varied the target cardinality. We generated 3 initial queries. The query marked as `Num` is a 3 table join query with 3 numeric range predicates, and original cardinality of approximately 600K. The query marked as `Cat` is a 3 table join query with 2 categorical predicates, each a part of a 3 level hierarchy. Its original cardinality is approximately 250K. The query marked as `Both` is a 3 table join query with 3 numeric and 2 categorical predicates and original cardinality approximately 125K. We varied the target cardinality from 1K (low selectivity) to 2M (high selectivity) tuples and plot the cardinality of the queries generated by our system with respect to the target cardinality. As can be seen, our technique generates queries that approximately satisfy the target cardinality constraints for a wide range of target and initial query cardinalities.

In our next experiment, we generated three random workloads of 50 queries each defined on the DMV database having numeric (`Num`), categorical (`Cat`), and both numeric and categorical (`Both`) predicates respectively. Each query in each workload is refined to target cardinalities of 10K, 100K...
and 1M tuples. We plot the average errors for each workload-target cardinality combination in Figure 5.5. As can be seen, the average errors are low except for the case of queries with categorical predicates only, and a low target cardinality (10K). This is primarily due to the underlying data distribution which prevents contraction to the target cardinality. For instance, consider a query with predicates \texttt{make = 'Porsche' and country = 'GM'}, and cardinality 186K tuples, which is to be contracted to 10K tuples. Given two categorical hierarchies (\texttt{make, model, color}) and (\texttt{country, state, city}), the best that any query refinement algorithm can do on the underlying DMV database is to contract the query to 40K tuples (\texttt{model = 'Carrera' and color = 'red' and city = 'Berlin' and state = 'Berlin'}). Further contraction of this query is not possible.
In our next experiment, we examine the effects of the size of the domains on which categorical predicates are defined. For the experiment shown in Figure 5.6, we generated 50 queries with 3 categorical predicates defined on domains with 4, 20 and 197 distinct values on the DMV database. Each query in the workload has a cardinality less than 100K. We invoke SnS to relax each query to target cardinalities: 100K, 200K and 300K tuples. We consider each predicate to form a single level hierarchy, resulting in $3! = 6$ possible navigation paths for query relaxation. Each query is relaxed by rolling up along every possible navigation path, with the final predicate selected using the subset sum procedure $\text{FinalPred}$ (Algorithm 11). In Figure 5.6, we plot the average error with respect to the domain size of the final predicate.

As can be seen from Figure 5.6, the average errors for small (4) and large (197) sized domains are higher. For small domains, this higher error is due to the fact that disjunctively adding or removing an additional predicate can significantly change the query cardinality. This leads to a coarse degree of control over the cardinality of the refined query. At the other extreme, for large domains, the $\text{FinalPred}$ procedure has a much finer degree of control on query cardinality. However attributes with a large number of distinct values often have many low frequency values, for which cardinality estimation may be difficult. This illustrates an interesting tradeoff for categorical domains, in which we need to balance the finer degree of control provided by large domains, with the associated higher relative errors in cardinality estimates for the many low frequency values present. We note that a similar tradeoff does not arise in numeric predicates, since one does not require cardinality estimates for each distinct value within a range, but for the range as whole.

![Figure 5.7: Varying dimensions and skew](image_url)

In our next experiment illustrated in Figure 5.7, we examine the effects of varying the number of predicates (dimensions), and the skew of the underlying data on the accuracy of our refinement procedures for numeric predicates. We generate workloads of 5 table joins defined on TPCH tables, with 2, 3, 5 and 7 numeric predicates, with each workload having 50 queries. The target
cardinality was fixed at 100K tuples. We plot the average error for each workload over both the uniform (Z=0) and skewed (Z=1) data. As can be seen, the errors are uniformly low (< 1%) and are not significantly affected by the number of dimensions or skew of the data.

5.6.2 Overheads

![Execution Times](image.png)

Figure 5.8: Execution Times

We measure the execution times of our technique by generating a workload of 10 queries on the DMV database, with each query having 3 numeric and 2 categorical hierarchical predicates. Each query was refined to 10K, 100K and 1M tuples. We measure the execution times on cold and warm caches. The execution time is the time to complete the entire refinement procedure, from the initial query specification to the final generation of the refined query. Figure 5.8 demonstrates that the average execution times are low (approx. 5s on cold caches) and independent of the target cardinality. This execution time is primarily concentrated on the execution of the $SSE$ procedure. Once the $SSE$ procedure instantiates the indexing structures, each round of the refinement process takes few ms to execute, demonstrating that our framework can support an interactive refinement interface with low response times.

5.7 Summary

*Stretch ’n’ Shrink* introduces a new way of handling the many/few answers problem often faced by database users in analytics. We observed that the means which database systems provide to limit result size (using the *stop-after* or *limit*) clause are different from the standard database model of having predicates on individual tuples. Consequently, we investigated the utility and feasibility of an interactive refinement framework that provides a means of controlling answer size by modifying query predicates. The core ideas on which SnS rests are:
5.7. **Summary**

- A search-space bounding procedure, that iteratively and interactively reduces the search space for refinements of the original query, and directly captures user preferences.

- A sampling based estimation procedure, which provides accurate estimates for multiple queries, enabling each round of SnS to be interactive.

Our experimental evaluation of an implementation of this framework in a real database system demonstrates the utility of our approach.
In the previous chapter, we looked at the problem of refining the predicates of a query in order to satisfy a cardinality constraint on its result. The problem was explored in the context of data analysis/business intelligence applications, with user preferences taken into account. In this chapter, we study an extension of the problem to multiple cardinality constraints on intermediate subexpressions. This is in the context of database testing, as described next.

A necessary step in the introduction of any new technology in a database system is to test its behaviour across a wide range of operating conditions. This often involves selecting a set of test databases, generating representative query workloads, and executing these workloads on the test databases to evaluate the effect of the new technology. The importance of testing and benchmarking has long been recognized in the database community and there are several standard benchmarks developed for various settings [4]. While these standard benchmarks serve as useful reference points, there is often a need to generate test databases that satisfy certain properties on (for instance) table size, column domains, skew on columns and correlation between columns. To this end there have been several efforts [51, 117, 20] for generating large amounts of synthetic data which satisfy required properties. Correspondingly, there exist tools [116, 112] that can generate a large number of valid SQL queries to execute on a test database. However, these tools take only the database schema as input and generate the queries without looking at the underlying data. Therefore, they cannot guarantee generation of queries with certain kinds of properties. In particular, we are interested in generating queries that satisfy cardinality constraints on intermediate subexpressions.

Consider, for instance, an improvement to a join algorithm being introduced in a database system. A natural step in evaluating the algorithm would be to test its performance across varying
sizes of inner and outer inputs. Given a fixed underlying test database, the input sizes can be controlled by varying the selection predicates on the base relations. However, in the absence of additional information, database testers currently have no means other than a cumbersome trial and error procedure to find a choice of selection predicates that result in the query satisfying the input size targets. To make the problem even more challenging, one might want to test how the join algorithm operates as a component in a pipelined query processing architecture. In this case, one would like to generate queries with varying intermediate join result sizes. This can be particularly difficult to do in the presence of skew and inter-attribute correlations.

Figure 6.1 describes a sample test case in this setting. We are given a query $Q$, a set of target cardinality constraints on intermediate subexpressions in the query evaluation plan, and a test database $D$ on which the query is to be executed. Our goal is to modify $Q$ to generate a new query $Q'$ that satisfies the target cardinality constraints when executed on database $D$. The class of modifications we consider in this chapter are modifications to the range selection predicates of $Q$. We refer to this problem as the Targeted Query Generation (TQG) problem.

In this chapter, we study the TQG problem, and provide a formal analysis as well as practical solutions. We extend previous hardness results for the problem established for the special case of a single cardinality constraint [21], and prove lowerbounds on approximating the problem in this case. Our analysis demonstrates the difficulty of the problem, and illustrates the error guarantees one can expect. Further, we analyze the general case of multiple target cardinality constraints and show how it is affected by statistical properties of the underlying database. In particular, we demonstrate that several cases of interest can be reduced to the problem of finding the best fit solution to a system of linear equations.

We follow up the analysis with a description of a practical algorithm that can be used to quickly generate queries that approximately satisfy the target cardinality constraints on subexpressions.
Our algorithm progressively refines the range selection predicates using a novel search procedure that utilizes sampling based techniques for fast and accurate cardinality estimation. We have prototyped our techniques inside the Postgresql database system, and demonstrate the utility of our solution through an extensive experimental evaluation.

The rest of the chapter is organized as follows. In Section 6.1 we describe related work. We then present a formal analysis of the targeted query generation problem in Section 6.2. We follow this with a description of our new algorithm in Section 6.3. We present a detailed experimental evaluation of our technique in Section 6.4 and conclude in Section 6.5. Before we proceed, we quickly review the differences between the problems considered in this chapter, and the previous one:

**Targeted Query Generation vs. Query Refinement**

The primary difference between the problems considered in this chapter and the previous one is that here we consider *multiple* cardinality constraints, while in query refinement only a single constraint is considered. A secondary, but still important difference is that in targeted query generation, we *do not* require user preferences on the specific values of the predicates. The only thing that matters is the cardinality of intermediate data, since the purpose of targeted query generation is to generate test queries. This is in contrast to query refinement, where user preferences are paramount, and consequently affects the design of the system.

### 6.1 Related Work

Bruno et al. [21] first investigated the TQG problem. Their work primarily considered the special case of TQG with only one cardinality constraint. They formally proved that this restricted version of the TQG problem is NP hard. They also introduced a heuristic hill climbing approach that assumes independence between the selectivities of the different predicates. In our present work, we establish formal guarantees on the hardness of approximating the problem, and describe a practical algorithm that avoids any assumptions about the statistical properties of the data.

The QAGen system [15] introduces a complementary approach towards the targeted testing problem. Instead of generating a test query given the test database, the approach generates a test database given the test query. To do so, QAGen introduces symbolic query processing which necessitates the use of constraint solvers to generate the underlying database. The primary drawback of the approach is that it generates a different database instance *for each test case*. As a result, the storage overheads of applying this approach for large scale testing of a new feature may be unacceptable. In addition, QAGen suffers from the overheads of using an expensive constraint solver which makes it unacceptably slow for large databases. For instance, the QAGen paper reports a database generation of approximately 20 hours for a test case based on TPC-H query 3 and a 1
GB database size. More than 80% of this time is spent in the constraint solver. Our experimental evaluation shows that our techniques are significantly faster. One advantage of QAGen is that it attempts to satisfy the test case exactly, while we attempt to approximately satisfy the test case. However, their exact approach comes with significant overheads, and we contend that in many cases, generating a query that approximately satisfies the test case should suffice.

The primary focus of research in database testing has been on the test database generation problem. Gray et al. [51] introduced techniques for generating large amounts of data having specific data distributions. More recently, the MUDD data generator [112] for TPC-DS [113] and the DGL language[20] introduced techniques for separating data distribution specification from the actual data generation. Our work enables targeted query generation without modifying the underlying database, and thus enables reuse of the standard test databases used at an organization.

The problem of query generation has received comparatively less attention, and typically, the query generation procedure is decoupled from the database generation process. Tools like RAGS [116] and QGen [112] enable large scale generation of valid SQL queries. Since these tools utilize only the schema and are independent of the underlying data, they cannot be used to generate test queries which guarantee specific properties during query execution. Our solution fundamentally differs from these techniques since it is data-aware. In particular, our solution accepts as input a fully specified SQL query, and therefore can utilize the output of such a query generation tool. More recently, Bati et al. [12] introduce a framework for generating test queries by modifying current test queries using execution feedback. Although similar in its basic principle, the focus of the paper is primarily on ensuring coverage of executor and optimizer code in the database system.

In addition to previous research on database testing, our work draws upon techniques for sampling based cardinality estimation of joins in relational databases [63, 61]. We extend these schemes using novel bounding arguments to avoid the cost of sampling multiple times from the disk.

6.2 Analysis

6.2.1 Problem Statement

We are given a database $D$ and a conjunctive query $Q$ defined on relations $R_1, \ldots, R_t$. The query has $d$ range predicates $P_1, \ldots, P_d$. Each range predicate takes the form of $R_j.x_i < C_i$ or $R_j.x_i > C_i$. We treat two-sided range predicates as two separate single sided predicates. In what follows, we assume without loss of generality that predicate $P_i$ is of the form $x_i < C_i$. Predicates of the form $x_i > C_i$ can be transformed into $-x_i < -C_i$. We use $C_i^{l}$ and $C_i^{u}$ to denote the lower and upper bounds of the domain on which $P_i$ is defined. Thus the constant $C_i$ in predicate $P_i$ is bounded as $C_i^{l} \leq C_i \leq C_i^{u}$, and the size of the domain of $P_i$ is $C_i^{u} - C_i^{l} + 1$. We refer to the domain of predicate $P_i$ as dimension $i$. The query $Q$ thus has $d$ dimensions.

Our framework can also by extended to equality predicates on non-numeric i.e categorical
domains by mapping the categorical domain to a numeric domain, and considering range predicates on the new numeric domain. In the final query, the ranges can be mapped back to obtain disjunctive equality predicates.

Range predicates can be modified only by changing the constant in the expression. Thus, for example age < 50 can be modified to age < 70 or age < 30, but not to age > 20. Queries can be modified by altering the range predicates. As in the previous chapter, we refer to this process of modifying queries as **query refinement**.

A **test case** τ is defined in terms of a query Q, a database D, and a set of m cardinality constraints N. Each cardinality constraint is defined in terms of a subexpression Qᵢ of Q, and a target cardinality Nᵢ for Qᵢ. We represent cardinality constraints as pairs (Qᵢ, Nᵢ). Likewise we represent test cases as triples (Q, D, N).

Given, these definitions, we now state the TQG problem

**Definition 8. Targeted Query Generation:** Given a test case τ : (Q, D, N), generate a test query Q' by refining Q such that Q', when executed on database D satisfies the cardinality constraints specified by N.

We use m to denote the number of cardinality constraints in the test case. We let d be the total number of predicates that can be refined to modify the cardinality of intermediate results i.e the number of dimensions of Q. We use n to represent the size of the largest domain on which a range predicate is defined. Our cost analysis is in terms of these three parameters.

**Example 14.** Consider the test case specified by Figure 6.1. We represent the cardinality constraints as (A, 100K), (A ⊗ B, 75K), (C, 125K). Therefore, m = 3. There are 4 range predicates (A.x < 1000, B.y > 500, C.z > 50 and C.w < 5000). Therefore, the number of dimensions d = 4. We set n to be the size of the largest domain among A.x, B.y, C.z and C.w.

An **evaluation layer** is defined as a module that given a query Q returns a cardinality estimate for Q. It can take the form of actual query execution, optimizer cardinality estimates, or any other cardinality estimation scheme. Our algorithms invoke the evaluation layer through the function CardEst which returns an estimate of the cardinality of the given query. Our techniques are independent of the evaluation layer in use. In the following analysis, we assume that the evaluation layer returns *exact* cardinality estimates i.e the error in the estimate is 0. Following [21], we measure the complexity of our algorithms in terms of the number of calls to the evaluation layer. Subsequently, in Section 6.3.3, we introduce an efficient sampling based evaluation layer for the practical realization of our algorithms. We now present a formal analysis of the hardness of the TQG problem. We first consider the special case when the number of cardinality constraints m is 1. We then consider the general case of m > 1 cardinality constraints.
6.2. Analysis

6.2.2 Single Cardinality Constraint

Consider the TQG problem given only a single cardinality constraint on the result size of the query Q. Since there is only one constraint, we drop the subscript, and denote it as (Q, N) where N is the target result cardinality. For this special case, Bruno et al. [21] proved the following Lemma:

Lemma 3. A lower bound on the number of calls to the evaluation layer to generate a query Q satisfying a single constraint (Q, N) with two parametric predicates $x_1 \leq C_1$ and $x_2 \leq C_2$ is $\Omega(n_{\min})$ where $n_{\min}$ is the minimum number of distinct values in $x_1$ and $x_2$. For $d > 2$ parametric predicates, there is a lowerbound of $\left(\frac{n+d-2}{d-1}\right)$ calls to the evaluation layer.

Note however, that this lowerbound would typically be too expensive since we would expect that $n >> d$, and therefore, even $O(n)$ calls to the evaluation layer might be considered unacceptable. We consider the hardness of determining an approximate solution to the TQG problem given a single constraint (Q, N). We use the absolute error as our metric in this setting. If we generate a query $Q^r$ which returns $N^r$ tuples, then the absolute error of $Q^r$ is $|N^r - N|$. We state lowerbounds for any algorithm that given a test case with a single constraint (Q, N) as input, guarantees returning a query $Q^r$ with absolute error less than a constant $E$. If such a query does not exist, the algorithm returns that no query is found. Our lowerbounds also carry over to any algorithm that guarantees absolute error less than $\epsilon N$ where $\epsilon$ is a constant and $\epsilon < 1$.

Lemma 4. Suppose there is an algorithm Alg that given a single constraint TQG problem (Q, N) guarantees a solution $Q^r$ with absolute error less than $E$ or $\epsilon N$ where $E$ and $\epsilon$ are constants and $\epsilon < 1$ (if such a solution exists). Then, the lowerbound on the number of calls to the evaluation layer made by Alg matches the lowerbound given by Lemma 3 for the exact solution.

Proof. We first prove for the case of two cardinality constraints $x_1 \leq C_1$ and $x_2 \leq C_2$. Our proof follows the same lines as the lowerbound proof given by Bruno et al. [21]. Consider a table with columns $x_1$ and $x_2$ where the domain of each column is $(1, \ldots, n)$. For a given vector $(v_1, \ldots, v_n)$, we generate a table that contains $E \times v_i$ tuples with value $(n - i + 1, i)$ for $(1 \leq i \leq n)$, and $E \times 2^{i+j-n}n - \alpha_{i,j}$ tuples with value $(i, j)$ where $1 \leq i \leq n$, $1 \leq j \leq n$, $i+j > n+1$, and $\alpha_{i,j}$ is the number of tuples $(i', j')$ such that $i' \leq i$, $j' \leq j$ and $(i', j') \neq (i, j)$. We can draw this table more easily as the following matrix, where entry $i, j$ is the number of points returned with $x_1 \leq i$ and $x_2 \leq j$.

\[
\begin{pmatrix}
0 & 0 & 0 & \ldots & E \times v_n \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & E \times v_3 & \ldots & 2^{n-2} \times E \times n \\
0 & E \times v_2 & 4E \times n & \ldots & 2^{n-1} \times E \times n \\
E \times v_1 & 4E \times n & 8E \times n & \ldots & 2^n \times E \times n
\end{pmatrix}
\]
Now, select any value of $N$ (target cardinality) between $E$ and $E_n$. Set $v_1, \ldots, v_n$ to be any values such that $E \times v_i \notin (N - E, N + E)$. The algorithm should return no match found with less than $n$ calls to the evaluation layer. Therefore, there must be some point on the diagonal $(i^*, i^*)$ which is not checked by the evaluation layer. Now we generate a new table which is identical to the previous one, except that $E \times v_{i^*} \in (N - E, N + E)$. The algorithm will evaluate as before, and return no match found even though there is a solution $(i^*, i^*)$. Therefore, the algorithm needs to call the evaluation layer for every point on the diagonal of the matrix.

To generalize to $d$ dimensions, notice that the proof for 2 dimensions hinges on a construction which allows us to fix a value on every element of the diagonal $(n - i + 1, i)$ for $1 \leq i \leq n$, while keeping all other elements independent. In $d$ dimensions, the equivalent construction fixed a value every element $(x_1, \ldots, x_d)$ which is a solution to the equation $x_1 + x_2 + \ldots + x_d = n + d - 1$. There are \((n^d + 2d)_{d-1}\) solutions to this equation. Therefore, we can extend the above adversarial construction to show that we must examine every one of these solutions.

A similar argument can be constructed to prove identical lowerbounds for the case when we would like to generate a query that satisfies the cardinality constraints with absolute error $\leq N\epsilon$. □

Lemma 4 states that any algorithm that provides absolute error guarantees in the form of $|N^r - N| < E$ or $|N^v - N| < \epsilon N$ has a cost lowerbound which matches the cost lowerbound for the exact solution to the problem. We now present procedure SingleConstraint that provides weaker data-dependent error guarantees while requiring just $O(d \log n)$ calls to the evaluation layer.

Algorithm 12 describes procedure SingleConstraint. The procedure accepts a query $Q$, and modifies its range predicates to generate a new query $Q^r$ that approximately satisfies the target cardinality constraint $N$. This is accomplished by iterating over predicates of $Q$ and refining them. For each predicate $P_i : x_i < C_i$ of $Q$, SingleConstraint invokes the function MaxTrans that performs a binary search between the lower $C_{i^l}$ and upper $C_{i^u}$ bounds of the domain of predicate $P_i$, returning a query that minimizes the absolute error. MaxTrans performs the same functionality as the function described in the Chapter 5 for computing maximal transformations (Algorithm 6). For each value of the predicate considered by the binary search, MaxTrans invokes the evaluation layer through function CardEst to get a cardinality estimate for the given query.

**Example 15.** Suppose our target cardinality is $3477$ tuples and our query is on a single table $R$ with $10000$ tuples. The query has just one predicate $R.x < C$ which can take values of $C$ between $1$ and $1001$, and the data is uniformly distributed on column $R.x$. Given this predicate, procedure MaxTrans will do a binary search between these bounds taking values for val as (in order): 501, 251, 376, $\ldots$, 349. This results in the predicate $R.x < 349$ which returns $3480$ tuples.

Let $M_i$ be the maximum frequency on dimension $i$ in the result of $Q$. Let $\epsilon$ be the minimum value of $M_i$ over all dimensions $i$ i.e $\epsilon = \min_i (M_i)$. We now state bounds on the error and performance of SingleConstraint
Algorithm 12 Single Constraint Procedure

1: Database $D$

2: $SingleConstraint(Query \; Q, \; Target \; N)$

3: $Q' = Q$

4: $E = Error(CardEst(Q'^{}, D), N)$  \text{Check Error of $Q$}

5: \textbf{for all} Dimensions $i$ \textbf{do}

6: \hspace{1em} $P = Predicates(Q)$

7: \hspace{1em} $P_i = NULL$  \text{Disable $i^{th}$ predicate}

8: \hspace{1em} $Q_P = Query(P)$  \text{Generate new query}

9: \hspace{1em} $Q' = MaxTrans(Q_P, N)$

10: \textbf{end for}

11: $E' = Error(CardEst(Q', D), N)$  \text{Check Error of $Q'$}

12: \textbf{if} $E' < E$ \textbf{then}

13: \hspace{1em} $Q'^{'} = Q'$

14: \hspace{1em} $E = E'$

15: \textbf{end if}  \text{If Error is reduced, replace query}

16: \textbf{end for}

17: return $Q'$

18: $MaxTrans(Query \; Q, \; Target \; N)$

19: $i = NullDimension(Q)$  \text{Disabled Dimension}

20: $min = C^l_i$  \text{Set min, and max}

21: $max = C^u_i$  \text{to domain boundaries}

22: $val = (min + max)/2$

23: \textbf{while} ($min \leq max$) \textbf{do}

24: \hspace{1em} $P_i = x_i < val$  \text{Set the predicate accordingly}

25: \hspace{1em} $Est = CardEst(Q, D);$  \text{Call Evaluation layer for Est.}

26: \hspace{1em} \textbf{if} $Est < N$ \textbf{then}

27: \hspace{1em} \hspace{1em} $min = val$

28: \hspace{1em} \textbf{else if} $(Est > N)$ \textbf{then}

29: \hspace{1em} \hspace{1em} $max = val$

30: \hspace{1em} \textbf{else}

31: \hspace{1em} \hspace{1em} return $Q$;

32: \hspace{1em} \textbf{end if}

33: \hspace{1em} \textbf{end while}  \text{End of loop for binary search}

34: return $Q$
Lemma 5. SingleConstraint guarantees an error of $e/2$ at a cost of $O(d \log n)$ calls to the evaluation layer.

Proof. SingleConstraint calls the procedure MaxTrans once for each dimension. Each call to MaxTrans makes $O(\log n)$ calls to the evaluation layer, giving us the overall cost of $O(d \log n)$. For the error guarantee, notice that we can add a straightforward condition to the binary search so that it stops at a value such that increasing or decreasing the value by 1 will increase the error. Therefore, a binary search of dimension $i$ can result in an error of at most $M_i/2$. Since we iterate over all dimensions, the global error is at most $\min_i(M_i/2) = e/2$.

The lowerbound on approximation given by Lemma 4 demonstrates that we can always come up with adversarial instances that make the targeted query generation problem hard to approximate. On the other hand, procedure SingleConstraint shows that a simple binary search procedure can easily guarantee an error bound of $e/2$ at a cost of $O(d \log n)$. We argue that on reasonable test databases, this would serve as a tight enough error bound. Note that this is a worst case error bound, and in practice the error is expected to be lower.

6.2.3 Multiple Constraints

The discussion in the previous section establishes the approximation guarantees for the TQG problem with a single constraint. We now analyze the case when we have multiple constraints $N : (Q_1, N_1), \ldots, (Q_m, N_m)$.

The TQG problem becomes significantly more challenging in the presence of multiple constraints. It is easy to observe that in this case, it is possible for the constraints to be inconsistent. As a simple example, consider a query $R \bowtie S$ with some selection predicates on $R$ and no predicates on $S$ and suppose every tuple of $R$ joins with 1 tuple of $S$. Let the test case be specified as $(R, N_R), (RS, N_{RS})$. If $N_R \neq N_{RS}$, one can easily see that the constraints are inconsistent. In general, while there might be several solutions for each of the cardinality constraints, it isn’t necessary that these sets of solutions intersect providing a solution that satisfies all $m$ constraints.

We now show how the hardness of the TQG problem with multiple constraints is affected by statistical assumptions on the data. We first introduce some notation which we require for this analysis. Let $f_i$ be the cumulative distribution function on dimension $i$ corresponding to predicate $P_i : x_i < C_i$. Therefore, $f_i(C_i^l) = 0$ and $f_i(C_i^u) = 1$. Let $V(Q_j)$ denote the set of dimensions included in subexpression $Q_j$. We denote the joint cumulative distribution function over $V(Q_j)$ as $f_{Q_j}$. Let $N_j^m$ be the cardinality of subexpression $Q_j$ when for every $i \in V(Q_j)$, we set $P_i : x_i < C_i^u$. Since each predicate is set to the upperbound of its domain, $N_j^m$ is the maximum cardinality of subexpression $Q_j$.

The TQG problem can then be considered to be the problem of identifying $C_1, C_2, \ldots, C_d$ such
that:

$$\forall 1 \leq j \leq m \prod_{i : i \in V(Q_j)} f_i = \frac{N_j}{N^m_j}$$

Each predicate $P_i$ can be set to $x_i < C_i$ to obtain the resulting test query.

**Independence assumptions**

To begin with, we assume independence between the columns of the table. We make these assumptions only to highlight a special case of the TQG problem which is amenable to analysis. Our general solution for the problem is described in Section 6.3.

With inter-column independence, we have, for each subexpression $Q_j$:

$$f_{Q_j} = \prod_{i : i \in V(Q_j)} f_i$$

The TQG problem then becomes

$$\forall 1 \leq j \leq m \prod_{i : i \in V(Q_j)} f_i(C_i) = \frac{N_j}{N^m_j}$$

Let $y_i = -\log f_i(C_i)$. Similarly, let $z_j = \log \frac{N^m_j}{N_j}$. We denote the $y_i$s using a $d$ dimensional vector $\mathbf{y}$, and the $z_j$s with a $m$ dimensional vector $\mathbf{z}$. Similarly, we can represent the $m$ sets $V(Q_j)$ as a $m \times d$ matrix $\mathbf{V}$. Then the TQG problem with inter-column independence becomes:

$$\forall 1 \leq j \leq m \sum_{i : i \in V(Q_j)} y_i = z_j$$

or $\mathbf{V}\mathbf{y} = \mathbf{z}$

We need to find a solution to this system which satisfies the additional constraints that $\forall_i y_i \geq 0$.

**Example 16.** Consider the test case defined in Figure 6.1 and assume that the columns are independent. Suppose that the maximum cardinality of $A$ is $200K$, of $A \bowtie B$ is $100K$ and of $C$ is $200K$.

We can represent the TQG problem as the following system of equations:

$$
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1
\end{pmatrix}
\begin{pmatrix}
y_{A,x} \\
y_{B,y} \\
y_{C,z} \\
y_{C,w}
\end{pmatrix} =
\begin{pmatrix}
\log 200K/100K \\
\log 100K/75K \\
\log 200K/125K
\end{pmatrix}
$$

Suppose we solve the system of equations and obtain values of $y_i$ for every dimension $i$. This in turn provides a value of $f_i(C_i)$, which corresponds to the selectivity of predicate $P_i$. However, we are actually interested in the value of $C_i$. Let $R$ be the relation on which predicate $P_i$ is defined. Then, the selectivity $s = f_i(C_i)$ implies a target cardinality of $s|R|$ for the query $\sigma_{P_i}R$. Since there
is just one constraint, we can apply the procedure \textit{SingleConstraint} on this resulting query and solve for $C_i$. The resulting predicate $P_i : x_i < C_i$ has the property that the query $\sigma_{P_i}R$ returns approximately $s|R|$ tuples. The approximation guarantee is defined as per Lemma 5.

Observe that in our test cases, each cardinality constraint is specified as an intermediate subexpression in a query evaluation plan. Therefore, cardinality constraints are defined on subsets of the relations on which the query executes. As a result, we do not require independence between the selection columns on the same table. Therefore, instead of having separate cdfs $f_i$ for each column $x_i$, we simply define a joint distribution function $f_R$ over the set of dimensions of table $R$. The system of equations that are formed as a result can be appropriately solved, following which we use procedure \textit{SingleConstraint} to obtain appropriate selection predicates for $R$. We do however require that the join column of $R$ be independent of the selection columns on $R$, which in turn implies inter-table independence.

\textbf{Numerical Solutions to equations}

Given the system of equations $Vy = z$ we seek solutions for $y$ subject to the constraint $y \geq 0$. The system of linear equations given by $Vy = z; y \geq 0$ can be under-determined, have a unique solution, or be overdetermined. Each of these cases is possible in the TQG problem as we illustrate in the following example:

\textbf{Example 17.} Consider a join of $R$ and $S$ with selection predicates on $R.x$ and $S.y$. With just one cardinality constraint on $R \bowtie S$, the corresponding system of equations has one equation and two variables, and is therefore under-determined. With cardinality constraints on both $R \bowtie S$ and $R$, we have two equations, and two variables, giving us a unique solution. Finally, if we have cardinality constraints on $R \bowtie S$, $R$ and $S$, we have three equations on the same two variables, leading to an overdetermined system of equations.

If the system of equations is consistent (i.e. has at least one solution), we can solve for it using standard techniques to obtain a solution for $y_i$. If there are multiple solutions, any of these solutions would work equally well. All that we require are values for $y_i$ that can be used to solve for $C_i$ to obtain the new predicates $P_i$.

If however, the system of equations is inconsistent, there is no solution $y$ that satisfies all the equations in the system. In this case, we seek solutions that optimize some norm between $Vy$ and $z$. Depending on desired semantics any of the candidate norms, such as $L_1$, $L_2$ and $L_\infty$ can be of interest. Consider the case of the $L_2$ norm. In this case the problem takes the following form:

$$\min \|Vy - z\|^2$$

s.t $y \geq 0$
This problem is amenable to a closed form numerical solution. Notice that the $L_2$ norm we are trying to minimize subject to linear constraints is a convex function. We apply primal-dual techniques to derive a solution. The dual of the problem can be derived as follows:

$$\min_y L(y, \lambda) = (Vy - z)^T(Vy - z) - \lambda^T y$$

Setting the derivative of $L(y, \lambda)$ w.r.t $y$ as zero we obtain the optimal values of $y$ minimizing $L(y, \lambda)$ as a function of $\lambda$. In particular we obtain:

$$y^* = \frac{1}{2}(V^TV)^{-1}(2z^TV + \lambda^T) \quad (6.1)$$

Substituting this choice of $y^*$ and taking the derivative w.r.t $\lambda$, we obtain the optimal value of $\lambda$. This can be substituted back in the original equation to obtain the $y$ vector as a solution to the $L_2$ minimization problem. A similar analysis can be conducted for the case of other norms such as $L_1$ and $L_\infty$. However in these cases one can show that no closed form numerical solution is available and one has to resort to numerical procedures to obtain the optimal vector $y$ [5].

We note that minimizing the $L_2$ norm is a particularly appropriate choice for the problem under consideration. Minimizing $||Vy - z||^2$ is equivalent to minimizing the following error function

$$E = \sum_{1 \leq j \leq m} (\log \frac{N_j^r}{N_j})^2 \quad (6.2)$$

where $N_j^r$ is the cardinality of the resulting query $Q^r$ at the intermediate subexpression $Q_j$. Essentially, minimizing the $L_2$ norm is equivalent to minimizing the sum squared logarithmic relative error over the set of cardinality constraints in the test case. This metric is does not have any bias towards subexpressions $Q_j$ with larger target cardinalities $N_j$, and equally penalizes overshooting and undershooting a constraint.

**General Case**

Our analysis in the previous sections shows that the TQG problem with multiple constraints is solvable if our test database guarantees the property that the join column is independent of the selection columns, and consequently that the selection columns across tables are independent. If we drop these assumptions as well, the TQG problem becomes less amenable to analysis since the data distribution of a selection column can be affected by the distribution on the join column.

Advances in data generation techniques have ensured that generating test databases with inter-table statistical dependencies is an easy task. As a consequence we still require a technique to generate targeted test queries which can handle such databases. In the next section, we describe our solution for the general case of this problem. The adversarial argument of Lemma 4 shows that one cannot expect to obtain an efficient algorithm that guarantees returning a query with an arbitrarily small bound on the error. Instead, we present a best-effort algorithm for the TQG problem. In Section 6.4, we present an experimental evaluation demonstrating the effectiveness of our algorithm.
6.3 The TQGen Algorithm

We now present the Targeted Query Generation (TQGen) algorithm for generating queries that satisfy cardinality constraints on intermediate subexpressions. TQGen takes as input a test case \( \tau = (Q, D, N) \) and generates a new query \( Q' \) that (approximately) satisfies the constraints \( N \) when executed on database \( D \). We note that the TQGen procedure does not make any independence assumptions between attributes. As demonstrated in the previous section however, the targeted query generation problem is computationally difficult. Therefore TQGen is a best-effort search procedure that utilizes heuristics to guide its search for queries satisfying the test case.

The space of all possible refinements of the original query \( Q \) can be described as a \( d \) dimensional space, with each dimension corresponding to a predicate of \( Q \). Each point in this space corresponds to a setting of the \( d \) predicates, and therefore describes a unique query. For instance, Figure 6.2 describes a 2-dimensional space for a query with predicates on \( x_1 \) and \( x_2 \). In the figure, the point \( C_1, C_2 \) describes the original query in this space. TQGen explores this space to search for test queries that approximately satisfy the test case \( \tau \). It does so by bounding the search space, and then exploring this restricted space.

In the bounding phase, which we describe in Section 6.3.1, TQGen restricts the search space for solutions. This is done by generating a new query \( Q^U \) that is a superset of the original query \( Q \). \( Q^U \) is generated by refining each predicate of \( P_i : x_i < C_i \) of \( Q \) to \( x_i < C^U_i \) where \( C_i \leq C^U_i \). We define the semantics of our space bounding operation further in Section 6.3.1. This phase is analogous to Phase 1 of SnS described in Section 5.3.

In the exploration phase, described in Section 6.3.2, TQGen explores the space \( \forall_i x_i \leq C^U_i \) for valid test queries \( Q' \). An exhaustive search of this space would however be extremely inefficient. Therefore, we partition the \( d \) dimensional space of predicates using a grid as illustrated in Figure 6.2. The grid is generated by performing an equi-width partitioning along each dimension. Each cell in the grid is a candidate for further exploration through repartitioning at a finer granularity. We define a cell scoring function to evaluate the utility of exploring a given cell, and select the \( b \)-highest scoring cells for further exploration. The algorithm is a recursive procedure that repeats
6.3. The TQGen Algorithm

The process of partitioning and scoring at finer levels of granularity for the selected cells. This recursive partitioning is illustrated in Figure 6.2 in which the 3 cells have been chosen for further repartitioning.

At all times during query execution, the TQGen algorithm maintains the lowest error query seen till then. We utilize as our error function the sum squared logarithmic relative error function defined in Equation 6.2. However, our algorithm is error function independent, and can utilize any monotonic error function i.e if the error for any constraint \((Q_j, N_j)\) increases, then the total error cannot decrease.

TQGen uses an evaluation layer module for estimating the cardinality of the potential test queries considered. Our algorithm is independent of the evaluation layer used. In Section 6.3.3, we describe a sampling based evaluation layer designed specifically for use with the TQGen approach. We now present the bounding and exploration phases of the TQGen algorithm.

### 6.3.1 Space Bounding

The first step of the TQGen algorithm consists of identifying reasonable bounds on the search space to be explored for test queries. Each predicate \(P_i : x_i < C_i\) can take any value for \(C_i\) between \(C_l^i\) and \(C_u^i\). Our goal is to find a value \(C^U_i\) such that \(C_l^i < C^U_i \leq C_u^i\), and \(C^U_i\) can serve as an upper bound on dimension \(i\) for our exploration procedure.

An upper bound \(C^U_i\) for a dimension \(i\) is strict if no test query \(Q^r\) that satisfies the test case can have a predicate \(P^r_i : x_i < C^r_i\) with \(C^r_i > C^U_i\). It is important to note here that seeking strict upper bounds may not lead to any reduction in the space. Consider the following example:

**Example 18.** Suppose there is a target cardinality constraint of 4000 tuples on a single relation \(R\) which contains 10000 tuples. We have two range predicates \(x < 50\) and \(y < 50\). The domain of both the predicates is \([1, 101]\), the data is uniformly distributed on both columns \(x,y\) and they are independent of each other. In this case, the predicates \(x < 41 \land y < 101\) and \(y < 41 \land x < 101\) would both serve as solutions to the test case, and therefore the strict upperbound for \(x\) and \(y\) is 101 which implies no reduction in the space.

Example 18 illustrates why strict semantics may not lead to any reduction in the search space for the exploration phase. Therefore, we adopt a weaker definition of an upperbound. A query \(Q^U\) is a valid upperbound if \(Q^U\) overshoots all the cardinality constraints specified by \(N\) when executed on the database \(D\). A query \(Q^U\) overshoots a constraint \((Q_i, N_i)\) if executing \(Q^U\) on \(D\) returns \(N^U_i \geq N_i\) tuples at the subexpression \(Q_i\).

Notice that in the process of refining a query \(Q\) to a test query \(Q^r\), we can transform predicates through contraction (e.g, \(age < 50\) refined to \(age < 40\)) or relaxation (e.g, \(cost < 100\) refined to \(cost < 150\)). Contracting some predicates may require us to relax other predicates. For instance,
Algorithm 13 Space Bounding

1: \textit{Bound}(Q)
2: \textbf{for all} Dimensions \(i\) \textbf{do}
3: \(C_{i}^L = C_i\)
4: \(C_{i}^U = C_i\)
5: \textbf{end for} \hspace{0.5cm} \textit{Initially set bounds to the original values}
6: \textbf{for all} Constraints \((Q_j,N_j)\) \textbf{do}
7: \textbf{ComputeBounds}(Q_j,N_j,lower)
8: \textbf{end for} \hspace{0.5cm} \textit{First compute lowerbounds iterating over constraints}
9: \textbf{for all} Dimensions \(i\) \textbf{do}
10: \(P_i : x_i < C_i^L\).
11: \textbf{end for} \hspace{0.5cm} \textit{Modify predicates generating} \(Q^L\)
12: \textbf{for all} Constraints \((Q_j,N_j)\) \textbf{do}
13: \textbf{ComputeBounds}(Q_j,N_j,upper)
14: \textbf{end for} \hspace{0.5cm} \textit{Repeat for upperbounds}
15: \textbf{ComputeBounds}(Q_j,N_j,which)
16: \textbf{for all} Dimensions \(i \in V(Q_j)\) \textbf{do}
17: \text{Temp} = \(P_i\)
18: \(P_i = \text{NULL}\)
19: \(Q' = \text{MaxTrans}(Q_j,N_j)\) \hspace{0.5cm} \textit{Calling MaxTrans to obtain new value}
20: \(C_i' = \text{Value}(i \in V(Q'))\)
21: \(P_i = \text{Temp}\)
22: \textbf{if} which == lower \textbf{then}
23: \textbf{if} \(C_i' < C_i^L\) \textbf{then}
24: \(C_i^L = C_i'\)
25: \textbf{end if} \hspace{0.5cm} \textit{Replace lowerbound if smaller}
26: \textbf{else}
27: \textbf{if} \(C_i' > C_i^U\) \textbf{then}
28: \(C_i^U = C_i'\)
29: \textbf{end if} \hspace{0.5cm} \textit{Replace upperbound if larger}
30: \textbf{end if}
31: \textbf{end for}

in Example 18, contracting the predicate \(y < 50\) to \(y < 41\) results in the relaxation of the predicate on \(x\) to \(x < 101\).

We would like our upperbound computation procedure to take into account the effects of pred-
icate contraction. As a result, we adopt a two-step approach to this problem. Given $Q$, we first compute a new query $Q^L$ with predicates $x_i < C^L_i$ that is guaranteed to undershoot all the constraints in $N$. Each predicate $x_i < C^L_i$ satisfies the following property with respect to $Q$:

**Property 3.** If we replace any predicate $x_i < C_i$ of $Q$ with $x_i < C^L_i$, and keep all other predicates as before, the resulting query undershoots all cardinality constraints $(Q_j, N_j)$ s.t. $i \in V(Q_j)$.

Essentially, $C^L_i$ acts as a lowerbound for dimension $i$ in the sense that if we only modify predicate $P_i$ to $x_i < C^L_i$ will lead to the resulting query undershooting all constraints affected by $P_i$. In other words, $C^L_i$ is the minimum of the maximal contractions (as defined in Chapter 5) along predicate $x_i$ over the set of constraints. We then generate $Q^U$ with predicates $x_i < C^U_i$ by refining the query $Q^L$. The predicates $x_i < C^U_i$ of $Q^U$ satisfy the following property with respect to $Q$:

**Property 4.** If we replace any predicate $x_i < C_i$ of $Q$ with $x_i < C^U_i$, and replace all other predicates $x_{i'} < C_{i'}$ with $x_{i'} < C^L_{i'}$, the resulting query either overshoots all cardinality constraints $(Q_j, N_j)$ s.t. $i \in V(Q_j)$, or $C^U_i = C^u_i$.

The resulting query $Q^U$ serves as a reasonable upperbound since it explicitly takes into account the effects of contracting some of the predicates. $Q^U$ is analogous to the bounding query defined for the query refinement problem in the previous chapter.

Procedure *Bound* presented in Algorithm 13 describes how we compute $Q^U$. *Bound* utilizes the *MaxTrans* procedure described in Algorithm 12. Recall that the *MaxTrans* procedure accepts a query $Q$ with $d$ predicates, a cardinality constraint $(Q, N)$ and a single predicate $P_i$ specified as NULL. Given this information, *MaxTrans* computes the value of predicate $P_i$ that minimizes the absolute error for the specified cardinality constraint $(Q_j, N_j)$.

For each constraint $(Q_j, N_j)$ and each predicate $P_i \in V(Q_j)$, our procedure calls *MaxTrans* to compute the best value of predicate $P_i$ which minimizes the error only for $(Q_j, N_j)$. By taking the minimum such value of $P_i$ across all constraints $(Q_j, N_j)$, we compute a lowerbound for the predicate $P_i$. This process is shown through lines 6-8 and 22-24 in Algorithm 13. Having then generated $Q^L$ (lines 9-11) by replacing $C_i$ with $C^L_i$, we repeat this process for upperbounds (lines 12-14), and take the maximum value of predicate $P_i$ returned by *MaxTrans* as an upperbound for dimension $i$.

### 6.3.2 Exploration

Given the query $Q^U$, we utilize procedure *Explore* described as Algorithm 14 to search for potential test queries $Q^r$ that minimize the error function $E$ given by Equation 6.2. *Explore* takes as input a cell, which is defined by setting upper and lower bounds on each dimension. Each cell is thus specified by two $d$ dimension vectors $(T^u, T^l)$. The initial cell is thus specified by the vectors $(C^U_1, \ldots, C^U_d)$ and $(C^L_1, \ldots, C^L_d)$ Given a $d$ dimensional vector $T$, we use $Q(T)$ to denote query $Q$.
Algorithm 14 Exploration of the space

1: Explore(Cell $T_u, T_l$)
2: If isEmpty() return
3: Partition $T_u, T_l$ into upto $k^d$ smaller cells.
4: List = NULL
5: for all $T_u^i, T_l^i$ ∈ Partition do
6: Compute Error $E_i$ for $Q(T_u^i)$ by calling CardEst
7: if $E_i < E_{best}$ then
8: Set $Q^r = Q(T_u^i)$
9: end if
10: Score The Cell defined by $T_u^i, T_l^i$
11: if List has space OR Score($T_u^i, T_l^i$) > Score(List) then
12: Add $T_u^i, T_l^i$ to list. Trim if necessary
13: Update Score(List)
14: end if
15: end for
16: for all $(T_u^i, T_l^i) \in$ List do
17: Explore($T_u^i, T_l^i$)
18: end for

with each predicate $P_i$ set to the corresponding value in $T$. For instance, $Q(3, 5, 7)$ would represent the query with predicates set as $x_1 < 3$, $x_2 < 5$, and $x_3 < 7$. Similarly given a constraint $(Q_j, N_j)$, $N_j(T)$ denotes the cardinality of subexpression $Q_j$ on execution of $Q(T)$.

Procedure Explore partitions each of the $d$ dimensions of a cell into at most $k$ segments. This partitioning is performed in an equi-width manner. This results in the given cell being decomposed into upto $k^d$ smaller cells. Since each dimension $i$ is partitioned into at most $k$ segments, there are at most $k + 1$ boundary values being considered for predicate $P_i$ at each step. The $k + 1$ boundaries along the $d$ dimensions together define $(k + 1)^d$ intersection points in the $d$ dimensional space. Each such point corresponds to a potential test query. We evaluate each such query using our error measure given by Equation 6.2 (Alg. 14 line 5). This evaluation is done using $m$ calls to the evaluation layer, once for each subexpression $(Q_j)$ with constraint $(Q_j, N_j)$. At all times during execution Explore maintains information about the query with minimum error seen until then.

In addition to evaluating the queries defined by the cell boundaries, Explore also needs to select $b$ cells to further repartition and explore at a finer granularity (Alg. 14 lines 9-13). To do so, it computes scores for each cell in the grid, and selects the $b$ highest scoring cells for further exploration. The scoring function quantifies the utility of further exploring a cell by repartitioning it. In addition, it utilizes pruning techniques to rule out exploration of non-promising cells. We
next describe both these techniques in more detail.

**Scoring Cells**

At each step of the Explore procedure, we consider up to \(k^d\) cells for further exploration. However, to avoid an exhaustive search, we utilize a scoring function to score the potential benefit of exploring the given cell. Our scoring function utilizes the following two parameters.

**Number of constraints bounded:** For each cell \((T^u_i, T^l_i)\), we count the number of cardinality constraints \((Q_j, N_j)\) that are bounded by the cell boundaries. We denote this set of bounded constraints by \(B(i)\). Therefore, we have \(j \in B(i)\) if \(N_j(T^u_i) \geq N_j \geq N_j(T^l_i)\). Thus \(|B(i)|\) is an upper bound on the number of constraints that can be satisfied by any query defined by predicates within the cell \(T^u_i\) and \(T^l_i\). Given two cells, \((T^u_1, T^l_1)\) and \((T^u_2, T^l_2)\), we assign a higher score to \((T^u_1, T^l_1)\) if \(|B(1)| > |B(2)|\).

**Uniformity across constraints:** Suppose two cells bound the same number of constraints, and we need to drop one of them. In this case, we can define the distance of the cardinality constraint \((Q_j, N_j)\) from the cell boundaries \((T^u_i, T^l_i)\) as:

\[
D_j(i) = \frac{N_j(T^u_i) - N_j}{N_j(T^u_i) - N_j(T^l_i)}
\]

Let \(\overline{D(i)}\) be the average distance \(D_j(i)\) for all \(j \in B(i)\). We then compute the standard deviation of these distances across the set of bounded constraints:

\[
\sigma^2(i) = \frac{1}{|B(i)|} \sum_{j \in B(i)} (D_j(i) - \overline{D(i)})^2
\]

We assign higher scores to cells with lower values of \(\sigma^2(i)\). The reasoning behind this choice is that we desire test queries that approximately satisfy all the constraints in the test case. Therefore, we prefer cells that are equidistant from the target cardinality constraints.

Together, these are the two components of our cell scoring function. We use the first component (number of constraints bounded) for the purposes of scoring, and use the second component to break ties.

**Pruning**

The scoring function described in the previous section enables our procedure to evaluate the utility of further exploring a cell. We now describe two pruning techniques that we use to reduce the costs of our search procedure.

**Lowerbound-Upperbound based pruning:** If the lowerbound \(T^u_i\) of a cell \((T^u_i, T^l_i)\) undershoots exactly \(m\) of the constraints, then the cell cannot bound more than \(m\) constraints i.e \(|B(i)| < m\). Consequently, if the last element in the current list of cells bounds more than \(m\) constraints, we can prune \((T^u_i, T^l_i)\) without evaluating the upperbound \(T^u_i\) of the cell.
Error based pruning: While processing a cell \((T^u_i, T^l_i)\), suppose that for a constraint \((Q_j, N_j)\), we have \(N_j(T^l_i) > N_j\) i.e the lowerbound of the cell overshoots the constraint. In this case, for constraint \((Q_j, N_j)\), no query in the cell \((T^u_i, T^l_i)\) can have an error lower than \((\log N_j(T^l_i)/N_j)^2\).

Likewise, suppose \(N_j(T^u_i) < N_j\) i.e the upperbound of the cell undershoots the constraint. In this case, no query in the cell can have an error lower than \((\log N_j(T^u_i)/N_j)^2\) for constraint \((Q_j, N_j)\).

As we process the constraints, for each constraint \((Q_j, N_j)\) that is not bounded by the cell i.e \(j \notin B(i)\), we add the error lowerbound described above to a running sum \(S_{Err}\). If at any point in this processing, this lowerbound sum \(S_{Err}\) exceeds the current best error \(E_{best}\), we can stop processing the cell and prune it from consideration.

Summary

To summarize, the \(TQGen\) procedure takes a \(d\) dimensional cell, partitions it into \(k^d\) smaller cells, and chooses upto \(b\) of them for further exploration. We select cells for further exploration using the scoring and pruning functions which evaluate the utility of a cell. Among all the queries evaluated, \(TQGen\) returns the one minimizing the error function given by Equation 6.2.

We now analyze the cost of our algorithm. Each invocation of the \(Explore\) procedure involves the evaluation of upto \(O(k^d)\) potential test queries. Each query evaluation involves \(m\) invocations of the evaluation layer, once for each subexpression \(Q_j\) on which a constraint is defined. Therefore the cost per call to \(Explore\) is \(O(mk^d)\). Now suppose the procedure branches into \(b\) child cells upto a depth \(l\). Beyond \(l\), it selects only one further child at each level i.e \(b = 1\) after a depth \(l\). The total depth of the tree is \(\log_k n \geq l\). The total number of invocations performed is therefore \(1 + b + \ldots + b^{l-1} + b^l(\log_k n - l) = O(b^l \log_k n)\). Therefore, the total cost of the procedure is \(O(b^l mk^d \log_k n)\), which has the essential property of being logarithmic in the size of the domain \(n\).

6.3.3 Evaluation Layer

In section 6.2, we defined an evaluation layer as a structure which returns cardinality estimates for queries submitted to it. Our procedures are independent of the actual evaluation layer used. If we desire perfect cardinality estimates, we could actually execute the queries on the database as suggested in [21]. Alternatively, we could make use of optimizer cardinality estimates, histograms [75], sample views [86], statistics on intermediate tables [19], and sampling from base tables [63, 61] to obtain cardinality estimates for our algorithm. In our analysis upto this point, we have defined the cost of our algorithms in terms of the number of calls to the evaluation layer. However, for our algorithms to be practical, we need a fast and accurate evaluation layer.

We now present an evaluation layer that is specifically tailored to interact with the \(TQGen\) procedure. We first provide background on the sampling based cardinality estimation scheme utilized by our layer. We then present our techniques for generating an evaluation layer for the
bounding and exploration phases of our algorithms.

Sampling Scheme

We apply the cardinality estimation scheme designed for SnS as described in Section 5.2.4 to the TQGen algorithm. Our estimation procedure relies on sampling over a superset of the set of queries for which cardinality estimates are required. We refer to each invocation of the sampling procedure as a sampling step. A naive sampling based cardinality estimation scheme would incur a cost of one sampling step for each call to the evaluation layer. In the following sections, we show how we utilize our space bounding techniques to minimize the number of sampling steps required for cardinality estimation.

Evaluation Layer for Bounding

In the bounding phase of the TQGen algorithm, we compute lowerbounds $C^L_i$ and upperbounds $C^U_i$ for each predicate $P_i$ in the original query. Recall that the Bound procedure presented as Algorithm 13 computes $C^L_i$ and $C^U_i$ by keeping all predicates other than $P_i$ fixed, and performing a binary search on the domain of $P_i$. This procedure makes $m \log n$ calls to the evaluation layer to compute an upperbound or lowerbound of a predicate. Hence, the overall cost of the procedure is $2md\log n$ invocations of the evaluation layer.

To avoid $2md\log n$ sampling steps, we utilize the fact that the result of the query, with predicate $P_i$ disabled is a superset of the result if we set $P_i$ to any value in its domain. This is illustrated in the following example:

**Example 19.** Consider a query $Q$ having a join condition $R \bowtie S$ with selection predicates $R.x < C_x$ and $S.y < C_y$. Note that the query $Q \subseteq \sigma_{s,y<C_y}(R \bowtie S)$ for any value of $C_x$. Similarly, $Q \subseteq \sigma_{r,x<C_x}(R \bowtie S)$ for any value $C_y$.

Let $Q(\bar{P}_i)$ denote the query $Q$ with predicate $P_i$ disabled. Therefore, $\forall_i Q \subseteq Q(\bar{P}_i)$. To compute the lowerbound $C^L_i$ for predicate $P_i$, we perform one sampling step for each subexpression $Q_j$ of $Q(\bar{P}_i)$ for which $i \in V(Q_j)$. We store the result of this sampling procedure in memory, and MaxTrans procedure using this in-memory result as an evaluation layer. Effectively, we reduce the number of sampling steps by a factor of $\log n$ since we sample only once for each cardinality constraint $(Q_j, N_j)$ and predicate $P_i$ pair. Our technique differs from traditional sampling based cardinality estimation techniques in that we apply the predicate after the join procedure.

We further reduce our sampling costs by a factor of $d$ by combining the separate sampling steps for each predicate into one. This is done by generating a query that returns tuples that satisfy at least $d - 1$ of the $d$ predicates. This is analogous to the extended query $Q^e$ from Chapter 5 on query refinement. Consider the query given in Example 19. The new query:

$$\sigma_{r.x<C_x \lor s.y<C_y}(R \bowtie S)$$
returns the union of the tuples returned by the two queries $\sigma_{R.x<C}(R \bowtie S)$ and $\sigma_{S.y<C}(R \bowtie S)$. Therefore, we combine $|V(Q_j)|$ sampling steps for each constraint $(Q_j, N_j)$ into one sampling step of a query that returns tuples that satisfy $|V(Q_j)| - 1$ of the predicates. To enable this additional optimization, we modify the join pipeline to return tuples that satisfy at least $|V(Q_j)| - 1$ of the $|V(Q_j)|$ predicates in the subexpression $Q_j$. We adopt additional optimization measures to limit the size of the sample result stored in memory. These measures include maintaining frequency counts instead of the actual data, and pruning away sample elements that are far away from our current estimate of the lowerbound.

We repeat the process described above for computing upperbounds $C_i^U$ as well. The result of these two optimizations is that we sample only twice for each cardinality constraint $(Q_j, N_j)$, once for lowerbounds, and once for upperbounds. As a result, we reduce the total number of sampling steps from $O(2md\log n)$ to just $O(2m)$ steps.

**Evaluation Layer for Exploration**

In the exploration phase of the TQGen algorithm, we explore a space defined by a query $Q^U_i$ i.e $\forall_i x_i < C^U_i$. Given a potential test query $Q(T)$, defined by a vector $T$, we utilize the evaluation layer to obtain cardinality estimates for each subexpression $Q_j$ on which a cardinality constraint $(Q_j, N_j)$ is defined. Observe that any such potential test query $Q(T)$ returns a subset of the tuples returned by $Q^U_i$. Therefore, we can perform one sampling step for each subexpression $Q_j$ of $Q^U_i$, and store the result in memory. This result can be utilized to estimate the cardinality $N_j(T)$ of any potential test query $Q(T)$ at subexpression $Q_j$. As a result, we perform only $m$ sampling steps, one for each cardinality constraint $(Q_j, N_j)$.

Given query $Q(T)$, we wish to estimate $N_j(T)$ for each constraint $(Q_j, N_j)$. We do so by counting the number of tuples in the in-memory join sample of $Q_j$ which are dominated by $T$ (i.e are smaller along all dimensions) and then scaling up. This step utilizes the in-memory quadtree data structure described in Section 5.4.1 for the SnS framework. The quadtree described there maintains counts of the number of tuples within each node, and consequently avoids exploring nodes that are completely dominated by the query point.

**6.4 Evaluation**

In this section, we describe our system for targeted query generation, and present the results of an experimental evaluation of our techniques.

**6.4.1 System Architecture**

We have implemented a system for targeted query generation that utilizes the TQGen algorithm. The system consists of an external program, which we term the Coordinator, and a modified version
6.4. Evaluation

![Diagram of System Architecture]

Figure 6.3: System Architecture

of the Postgresql 8.0 database system. This architecture is illustrated in Figure 6.3. The coordinator accepts a test case specified as \((Q, D, N)\). For each cardinality constraint \((Q_j, N_j) \in N\), it submits a query \(Q_j\) to the database system. We have modified the execution engine of the Postgresql database system to provide the functionality of an evaluation layer for both the Bounding and Exploration phases of the TQGen algorithm as described in Section 6.3.3. Therefore, the query execution process for each subexpression \(Q_j\) acts as an evaluation layer for that subexpression. The coordinator communicates with these \(m\) processes through network sockets. Each evaluation layer process is unaware of the other processes, and only communicates with the coordinator. This design ensures that most of the complexity of our algorithm is kept outside the database engine, while leveraging the query processing primitives inside the engine.

6.4.2 Experiments

We now present the results of an experimental evaluation of our techniques. Our experiments were conducted using two TPC-H databases of size 1 GB each. One of them is the standard TPC-H database generated as per the benchmark specification. Since this database consists of uniformly distributed data, we also generated a TPC-H database with zipfian skew \(Z = 1\) using a publicly available tool [32]. The sizes of the tables and the symbols representing the tables used in our experimental evaluation are provided in Table 6.1. In our graphs, we refer to the database with skew as \(TPCH Z=1\), and the uniform database as \(TPCH Z=0\).

We first present our experiments evaluating the accuracy of our test query generation techniques. In Section 6.4.2, we evaluate our solution for the TQG problem with a single cardinality constraint, as described in Section 6.2.2. We follow this with an experimental validation of our analytical model for the TQG problem with multiple constraints on independent columns, as presented in Section 6.2.3. Finally, we evaluate the TQGen algorithm using workloads with varying numbers of
joins and cardinality constraints in Section 6.4.2. We complement the accuracy evaluation, with experiments presenting the execution costs of our approach in Section 6.4.2.

All our experiments utilize a sampling based evaluation layer as introduced in Section 6.3.3. For the bounding phase of the evaluation layer, described in Section 6.3.3, we stop the sampling step when we have 1000 tuples for each predicate. For the quadtree based evaluation layer for exploration as presented in Section 6.3.3, we halt the sampling step when either the quadtree has 5000 elements or a 5% random sample of the outer relation has been read. The cardinality estimation errors due to our evaluation layer were typically less than 1% and did not affect the quality of the queries generated by our system.

![Figure 6.4: Single Table Accuracy Experiments](image)

(a) Single Constraint: 1 Table  
(b) Single Constraint: 3 Table

**Single Constraint Case**

Figures 6.4(a) and (b) present our accuracy experiments for the TQG problem with a single constraint. For the experiment in Figure 6.4(a), we defined test cases on a selection query on table `PartSupp (PS)` with range predicates on attributes `availqty` and `supplycost`. The test cases were defined with target result cardinality ranging from 80K to 720K tuples i.e target selectivity from 0.1 to 0.9. We plot the relative error of our generated test queries, defined as \( \frac{N^r}{N} \) where \( N \) is the target cardinality, and \( N^r \) is the cardinality of the test query generated by our system. Figure 6.4(b)
presents a similar experiment with cardinality constraints defined on the result size of a three table query $L \bowtie O \bowtie C$ with range predicates on $C.acctbal$, $O.totalprice$, $L.extendedprice$ and $L.quantity$. As with the previous experiment, the target selectivity was varied from 0.1 to 0.9. Both the experiments were conducted on TPCH databases with $Z = 0$ and $Z = 1$. As the figure illustrates, the relative error of our technique is consistently within $1 \pm 0.02$. The only point outside this range is in Figure 6.4(a) for the $Z = 1$ database with selectivity 0.1. This point has a relative error of 1.03, and an absolute error of 2539 tuples from the target cardinality. Recall that the error guarantee of SingleConstraint is $e/2$ where $e$ is the minimum of the maximum frequencies along the dimensions of the query. In this particular example, $e/2$ is approximately 8000 tuples, which is much greater than the error at the data point.
Multiple Constraints: Analytical Model

We next validate our analytical model for the TQG problem with multiple constraints and independence between predicates as described in Section 6.2.3. We define test cases on three table query $PS \bowtie P \bowtie S$, with range predicates on $PS.availqty$, $PS.supplycost$, $P.retailprice$, and $S.acctbal$. This query was selected since the TPC-H specification does not define correlations between these three tables. We define the test cases with cardinality constraints on $PS$, $PS \bowtie P$ and $PS \bowtie P \bowtie S$. The solution to the corresponding system of equations implied a selectivity of $s$ each for $PS$, $P$ and $S$, where $s$ varied from 0.1 to 0.9. We computed values for the predicates on each of these relations utilizing procedure $SingleConstraint$ (Alg 12).

Figures 6.5(a) and (b) show the results of evaluating our generated test queries for the TPC-H databases with $Z = 0$ and $Z = 1$ plotted against the selectivity $s$. The labels $Upper$, $Middle$ and $Lower$ denote the subexpressions $PS \bowtie P \bowtie S$, $PS \bowtie P$ and $PS$, by virtue of their positions in the join pipeline. As can be seen, the generated queries validate our analytical model, and the relative error decreases as the target cardinality increases. We note that the higher relative errors at selectivity 0.1 for the $Upper$ graph in both figures are caused due to the fact that the target cardinality at this node is just 800 tuples (due to a combined selectivity of $0.1 \times 0.1 \times 0.1$) with respect to the outer relation size. Another interesting observation is that the relative error for the $Upper$ graph in Figure 6.5(a) increases when we increase the target selectivity 0.1 to 0.2. This is an effect of the fact that our generated test queries $undershoot$ the constraints at the $Middle$ and $Lower$ nodes for selectivity 0.1, and $overshoot$ these constraints for selectivity 0.2. In the first case, the errors partially cancel each other, while in the second case, they add up. Finally, we note that the errors for $Z = 0$ are typically lower than those for $Z = 1$. This is because columns with more skew have higher maximum frequencies, and the error guarantee of our $SingleConstraint$ procedure is defined in terms of these maximum frequencies.

Multiple Constraints: TQGen

In the previous two sections, we have evaluated our solution for the special case of TQG with single constraints, and validated our analytical model for multiple constraints with independent predicates. In this section, we evaluate our general purpose $TQGen$ algorithm. Recall that the $TQGen$ exploration phase constructs a grid by partitioning each dimension of a cell into $k$ equi-width partitions. It then selects $b$ cells for further repartitioning. In the following experiments, we set these parameters as $k = 3$, and $b = 2$upto a depth $l = 2$ in the recursion. Beyond depth $l$, we set $b = 1$.

We first present our results for the $TQGen$ algorithm given the query $P \bowtie PS$ with two range predicates on each of the relations. We fix the target cardinality of $PS$ to 500K tuples and vary the target selectivity of the join expression $P \bowtie PS$ between 0.1 and 0.9. Figures 6.6(a) and (b)
present our results for these test cases for TPC-H databases with $Z = 0$ and $Z = 1$. The queries generated by our technique have relative error bounded by $1 \pm 0.04$ for the upper and $1 \pm 0.02$ for the lower constraint. This experiment illustrates that the $TQGen$ algorithm can generate accurate test queries without utilizing the information that the selection columns of $P$ and $PS$ are independent.

![Figure 6.7: Varying parameters](image)

(a) Varying # of tables

(b) Varying # of constraints

We then evaluate the accuracy of the $TQGen$ algorithm for queries with varying numbers of tables and cardinality constraints. Each instance presented in Figure 6.7(a), describes a join query over $n$ tables with $n$ cardinality constraints, where $n$ varies from 3 to 5. The constraints are defined at the outer relation, and at the intermediate nodes in a left deep join tree. For each such instance, we generate 10 test cases, keeping the query fixed, and varying the target cardinality constraints. We execute $TQGen$ for each of these test cases, and calculate the relative error incurred by the generated test query at each constraint. We compute the average relative error over the constraints in a test case, and report the average of these errors over all 10 test cases in Figure 6.7(a). The results show that increasing the number of tables and constraints in the test case leads to an increase in the average error. This is due to the multi-objective nature of the TQG problem. However, we note that in all cases, the errors remain low and yield meaningful solutions to the TQG problem.

We also evaluate our techniques by fixing the number of tables in our test case, and varying the number of constraints. Figure 6.7(b) shows our results for a query on $P \bowtie PS \bowtie S$. We vary the number of subexpressions with cardinality constraints from 2 ($PS, P \bowtie PS \bowtie S$) to 5. As expected, the average error increases with the number of constraints. More interestingly, we note that the test cases with 4 or 5 constraints are overdetermined. For instance, the test cases with 5 constraints, have target cardinalities on $P$, $S$, $PS$, $PS \bowtie S$ and $PS \bowtie S \bowtie P$. This effectively specifies a constraint on every intermediate subexpression of a pipelined query evaluation plan. Even in this overdetermined case, our techniques generate queries that approximately satisfy the test case, albeit with higher error.
Efficiency

![Graph showing cache effects and partitioning effects](image)

**Figure 6.8: Execution Times**

We now describe the results of a performance evaluation of our algorithm. We conducted our experiments on a machine running Suse Linux with 4 GB memory and 3.6GHz clock speed.

We generated a set of test cases consisting of joins of \( n \) relations with \( n \) cardinality constraints, and \( n + 1 \) range predicates, with \( n \) varying from 2 to 5. These are marked as \((\text{Constraints, Dimensions})\) in Figure 6.8(a). We set the `lineitem` (\( L \)) as our outer relation. As can be seen from Figure 6.8(a), the execution time of our framework increases as we add more constraints and range predicates. Although, this experiment was run with parameters \( k = 3, b = 2 \) and \( l = 2 \), similar behaviour is observed with other choices of these parameters as well. We observed that with a cold cache, a significant chunk of the execution time is spent constructing the evaluation layer. This step involves executing an index nested-loops join pipeline, with a precomputed random sample at the outer relation, and the indices of the inner relations. When we repeated the experiment with a warm cache, we can observe that the execution times are significantly reduced. Typically, we expect multiple test cases to be provided to our system, and therefore, the process of generating the first test query should pave the cache for the other queries. Since our system relies only on samples and indices, it does not process much disk data, and as a result, it can take advantage of the large memories of modern machines.

The next experiment was performed on a warm cache to explicitly quantify the other factors that affect the execution time of the \( TQGen \) algorithm. In the experiment shown in Figure 6.8(b), we fixed the number of constraints to 2, and varied the number of predicates between 2 and 7 for a query involving a join of 5 tables. We varied the parameters \( k \) and \( l \), while keeping \( b \) fixed as 2. Figure 6.8(b) demonstrates that the expected execution time increases with the number of dimensions \( d \) and the parameter \( k \). Since our exploration procedure partitions a \( d \)-dimensional space using \( k \) buckets along each dimension, this time increase is expected. Alternate forms of
6.5. Conclusions and Future Directions

In this chapter, we have investigated several aspects of the Targeted Query Generation problem. We have formally established the hardness of obtaining algorithms with approximation guarantees, and identified cases where the problem is amenable to a solution. We have introduced new best-effort algorithms that utilize novel sampling and space bounding techniques to identify test queries. Our experimental evaluation shows the utility of our techniques, and demonstrates that despite the difficulty of the problem, it is possible to generate targeted queries for reasonable test databases.

An important future direction under consideration is the problem of generating multiple queries to satisfy multiple test cases in a batched fashion. One possible means of handling this scenario would be to combine the space bounding phases, while separating the exploration phases for the test cases. Another important direction to consider is the problem of minimally transforming the underlying test database to minimize the cardinality errors in the queries generated by $TQGen$. As a first step towards this problem, we explore the problem data generation given multiple test queries and intermediate cardinality and distributional constraints in the next chapter.
TDGen: Targeted Database Generation

In the previous chapter, we studied the problem of generating a test query that satisfies a set of cardinality constraints when executed on a given test database. In this chapter, we explore the problem of generating a database that satisfies a given set of query test cases. We consider test cases that specify a query and an associated set of cardinality and distributional constraints on intermediate subexpressions. As in the previous chapter, such cases enable stress testing of the behaviour and functionality of components in a database system under different operating conditions.

For example, consider a new implementation of an aggregation (Group-by) operator in a database system. Testing the performance of the operator requires executing it on various input sizes. Moreover, the performance of the aggregation operator also depends on the number of groups, and the distribution of group sizes. Additionally, measuring the performance of the operator as a part of a query plan requires testing across varying intermediate relation sizes and data distributions. Satisfying such test requirements is a challenging task for which there exist few supporting tools.

Recently, there has been a spurt of research towards the problem of satisfying test cases that specify runtime properties. Such research has taken the form of database-aware query generation frameworks such as [21] and TQGen [106] (Chapter 6), and query-aware database generation techniques [15, 14]. However, we note that the techniques for database-aware query generation [21, 106] cannot handle test cases that specify constraints on the data distribution at intermediate subexpressions, and provide only best-effort results for cardinality constraints, with weak guarantees of correctness. On the other hand, the proposed techniques for query-aware data generation [15, 14],
while providing limited support for distributional constraints, generate a separate test database for each test query. Moreover, they utilize expensive constraint solvers, making their performance orders of magnitude slower than the best query-independent database generation techniques [51, 20].

In this chapter, we study the problem of generating a single test database satisfying multiple test cases defined by queries associated with cardinality and distributional constraints on intermediate subexpressions. In accordance with the previous chapter, we term this problem the Targeted Database Generation (TDG) problem. We formally analyze the TDG problem for queries with selections, projections, joins, and aggregations, and prove hardness results for the associated cases. In addition to the formal analysis establishing the complexity of the problem, we describe a practical tool (TDGen) for targeted database generation. TDGen incorporates procedures for efficiently generating test data satisfying the cardinality constraints, while providing best-effort results for distributional constraints. In contrast to previous work [15], TDGen eschews expensive constraint solvers, and relies primarily on lightweight greedy algorithms, with no operation more expensive than sorting.

The rest of the chapter is organized as follows. In Section 7.1 we review related work. We formally describe our problem statement in Section 7.2, and follow it with an analysis of the complexity of the TDG problem in Section 7.3. We describe the TDGen framework in Section 7.4 and subsequently describe an extensive experimental evaluation of a prototype implementation of TDGen in Section 7.5.

7.1 Related Work

The system most similar in scope to TDGen is the QAGen system [15]. The QAGen system takes as input a test query annotated with cardinality constraints on intermediate subexpressions, and generates a test database such that the query when executed on the database satisfies the associated constraints. QAGen also supports distributional constraints to a limited extent, primarily for a subset of the join operations. In contrast to QAGen, TDGen supports multiple test queries simultaneously, generating a single associated test database. Moreover, TDGen avoids expensive constraint solvers deployed by QAGen, instead utilizing lightweight linear equation solvers, and as a consequence is considerably more efficient. On related lines, the SPQR system [14] is designed for the related problem of generating a database, given a query and its output. TDGen also incorporates a top-down data instantiation framework like SPQR, applying it in the context of the targeted database generation problem.

A related approach towards the problem of satisfying test cases specifying runtime properties is the approach of generating queries that take into account the underlying database. In particular, this approach focuses on cardinality constraints on intermediate subexpressions. However, Bruno et al. [21] proved that this problem is NP hard, and introduced a heuristic hill climbing approach for
data generation. In follow up work, the *TQGen* framework described in Chapter 6 provided formal guarantees on the hardness of approximation for the problem, and described a best-effort framework for query generation towards approximately satisfying the cardinality constraints. Our present work on the TDG problem inverts this approach, and ensures satisfying cardinality constraints on the data for a large class of test queries.

*TDGen* builds upon previous efforts at generating synthetic test data satisfying statistical properties. Gray et al. [51] introduced techniques for generating large amounts of data having specific data distributions. *TDGen* incorporates the data distribution generators described in [51]. Likewise, Bruno and Chaudhuri [20] define a Data Generation Language (DGL) for flexible specification of data properties. *TDGen* builds upon the iterator based approach towards flexible data generation espoused in [20], while taking as input a declarative specification of multiple test queries, and associated constraints. Similar efforts such as the MUDD data generator [112] for TPC-DS [113] also introduce techniques for declaratively specifying the distributional constraints on base tables. *TDGen* takes these ideas one step further, supporting distributional constraints on intermediate subexpressions as well.

The problem of query generation has received comparatively less attention, and typically, the query generation procedure is decoupled from the database generation process. Tools like RAGS [116] and QGen [112] enable large scale generation of valid SQL queries. RAGS [116] focuses on enhancing the coverage of system functionality. QGen [112] in contrast, focuses on randomly generating thousands of benchmark queries. Our present work on *TDGen* complements these techniques by providing a means to generate the underlying test data such that the test queries satisfy runtime properties. In the absence of tools like *TDGen*, many such randomly generated benchmark queries are likely to return empty results due to the properties of the data.

In recent work, Olston et al. [110] studied the problem of generating example data for dataflow programs e.g. the Map-Reduce programming paradigm [38]. The objective of their work is to generate an example data set such that the data satisfies the properties of (a) Realism i.e It is similar to a real world dataset (b) Completeness i.e it illustrates the semantics of the operator, and (c) Conciseness i.e the dataset is as small as possible. The objective is to help users understand the behaviour and correctness of their program without paying the cost of executing it on a large real-world dataset. Our present work, in contrast, focuses on enabling users understand the performance of the query processor by generating test data that satisfies runtime statistical properties. We note that our work can also be applied to the realm of dataflow programs, which have a similar structure to relational query evaluation plans.
7.2 Model

In this section, we formally present our problem statement and describe the scope of test cases considered in this chapter. We then introduce notation and terminology used throughout the chapter.

7.2.1 Problem Statement

A testing plan $\tau$ consists of multiple test cases generated from a fixed query template $Q$. A query template $Q$ is defined by a logical operator tree $T$, which completely defines the query, except for the constants in the selection predicates. For example, Figure 7.1(a) defines a query template specifying a join of two relations $A(x, i, g)$ and $B(y, z, i)$ on $A.i = B.i$, with a group-by on column $A.g$. However, the selection predicates $A.x > ?$, $B.y < ?$, $B.z < ?$ are not instantiated.

Each test case $(Q_i, C_i)$ is defined by a query $Q_i$ that is an instantiation of the query template $Q$, and a set of annotations $C_i$ specifying cardinality and distributional constraints on the edges of the logical operator tree. For example, Figures 7.1 (b) and (c) specify two test cases generated from the template in Figure 7.1(a). Each edge of the operator tree specifying the test case is annotated with an edge cardinality: for example Figure 7.1(b) specifies that the output of the selection operator on $A$ has a cardinality of 25K tuples. In addition to cardinality constraints, each edge of a test case can also be annotated with distributional constraint $F_r$ on an attribute $x_r$. Examples of such distributional constraints include the uniform distribution with $k$ distinct values ($\text{Unif}(k)$), a Zipfian Distribution with skew $x$ ($\text{Z}(x)$), or even a unique constraint ($\text{key}$).

We now formally state the Targeted Database Generation problem:

**Definition 9. Targeted Database Generation:** Given a testing plan $\tau$ consisting of $k$ test cases $(Q_1, C_1), \ldots, (Q_k, C_k)$ such that all the queries satisfy the same query template $Q$ i.e $\forall_i Q_i \in Q$, generate a single database $D$ such that for each test case $(Q_i, C_i)$, executing $Q_i$ on $D$ satisfies the constraints expressed by $C_i$. 

![Figure 7.1: Testing Plan](image)
7.2.2 Problem Scope

Given the definitions above, we now define the scope of testing plans considered in this chapter. Specifically, we focus on testing plans defining queries with Select, Project, Join and Aggregation (SPJA) operators with the following restrictions:

**Relevant Attributes:** Consider an edge $E$ in the logical operator tree directed towards operator $O$. We permit a distributional constraint $F_r$ on an attribute $x_r$ only if $x_r$ is relevant to the processing at $O$. A relevant attribute is defined as per the following rules:

- If $O$ is a selection operator, then the relevant attributes are the attributes on which the selection predicate is defined. For instance, in Figures 7.1(b) and (c), the distribution of $B.y$ and $B.z$ is specified at the edge between the scan operator on $B$ and the selection operator $B.y < ?, B.z < ?$.

- If $O$ is a join operator, then the attributes on which the join condition are defined are relevant. In Figure 7.1 $A.i$ and $B.i$ are relevant attributes for the join operator $A.i = B.i$

- If $O$ is an aggregation operator, then the relevant attributes are the grouping columns for the operator. In Figure 7.1 $A.g$ is the relevant attribute for the output of the join.

The focus on relevant attributes is a consequence of the envisaged applications of the TDG problem. For example, if we are testing the performance of a join operator, then the data distribution on a non-join attribute does not affect the performance of the operator.

**One operator per attribute:** Each attribute is relevant to at most one operator: either a select, or a join or an aggregation. This precludes queries where the same attribute $A.x$ has a selection condition $A.x < 100$, and multiple join conditions $A.x = B.x, A.x = C.x$. This ensures that each attribute is relevant at at-most one edge. As a result, we need to consider the data distribution $F_r$ of the attribute $x_r$ at at-most one intermediate subexpression of the query plan.

**Plan Structure:** We consider SPJA queries such that (i) The selection predicates can be pushed down to base tables (ii) There is at most one aggregation operator, which is at the top of the logical operator tree, and (iii) The joins have a single equality condition, with at least one of the join attributes having a uniqueness constraint (i.e it is a key). This captures a broad class of typical usage scenarios of database systems. We defer extensions of this basic framework to capture other operators and more complex query plans to future work.

7.2.3 Terminology

We now introduce terminology to be used through the chapter.

**Operators:** We denote operators of the logical operator tree by the symbol $O$. Each operator has up to two children, a left child $O_l$ and a right child $O_r$. If an operator has only one child, it is set as its right child $O_r$. Each operator $O$ has at most one parent $O_p$ which processes its output.
Chapter 7. TDGen: Targeted Database Generation

For a join operator $O$, we assume without loss of generality that the key input is its left child $O_l$, while the right child $O_r$ can have a general distributional constraint on the join attribute.

Finally, given a query $Q_i$, we represent the subexpression of $Q_i$ at an operator $O$ as $Q_i(O)$. Likewise, we denote the set of relations included in the subtree rooted at $O$ as $O.tables$.

**Attributes:** Let $O.Atts$ be the set of attributes contained in the tables over which the subexpression $Q_i(O)$ is defined. We let $O.relAtts$ be the set of relevant attributes at an operator $O$. Likewise, $O.parAtts$ is the set of output attributes at an operator that are relevant for some operator higher up in the tree. More formally if $O$ is the topmost operator, $O.parAtts$ is the empty set. For any other operator in the plan, we have:

$$O.parAtts = \{O_p.parAtts \cup O_p.relAtts\} \cap O.Atts$$

i.e it is the subset of relevant attributes at operators higher up in the plan, that are propagated through the current operator.

**Buckets:** Consider the set of selection predicates on an attribute $x_j$. Together these predicates split the domain of $x_j$ into $b(j)$ buckets $B(1, x_j), \ldots, B(b(j), x_j)$. The set of buckets are a minimal partitioning of the domain of the attribute such that each selection predicate $P$ along $x_j$ in a test query can be represented by a union of buckets $\cup P B(i, x_j)$.

**Cells:** Consider an operator $O$ in the logical operator tree, which is rooted over a subtree over the set of tables $O.tables$. Suppose there are $d$ selection attributes $x_1, \ldots, x_d$ over the set of tables $O.tables$. The buckets along the $d$ selection attributes partition the output of $O$ into a grid of $b(1) \times b(2) \times \ldots \times b(d)$ cells. Each subexpression $Q_i(O)$ can be represented as a union of multiple cells.

**Example 20.** Consider Figure 7.2. The two test queries have predicates $(x < 100, y < 50)$,
(x < 50, y < 100). The predicates along the dimensions partition the domain of x and y into 3 buckets, and therefore the entire relation is split into 9 cells. Each query can be represented as the union of multiple cells. For instance the query (x < 50, y < 100) corresponds to cells R_{11}, and R_{12}.

At a base table, let cell(B) denote the set of cells that are included in a Bucket B. For instance in Figure 7.2 cell(B(1, x)) = \{R_{11}, R_{12}, R_{13}\}. Similarly, let cell(Q_i(O)) denote the set of cells included in a subexpression Q_i(O). For ease of notation, we let the set of cells on the final output of a query Q_i be denoted by cell(Q_i). Likewise, for a table A, let cell(A) denote the entire set of cells at the output of the scan operation on A.

Consider a cell R at the output of a join operator O. There exist unique cells R_l and R_r in outputs of O_l and O_r such that R = R_l \bowtie O R_r. R_l and R_r are consequently the left and right child or R. Likewise, R \in parents(R_l) and R \in parents(R_r).

7.3 Analysis

In this section, we formally present the variations of the TDG problem for selections, aggregations and joins, and analyze the complexity of the problems.

7.3.1 Selection Operators

We first consider the simple case of multiple test cases being defined on a single relation T with size |T|. We first consider the case of only range selection predicates. Each test case \((Q_i, C_i)\) is defined by a set of \(d\) selection predicates \(L_1^i < x_1 < U_1^i, \ldots, L_d^i < x_d < U_d^i\).

An assignment \(A_j\) maps a frequency distribution \(F_j\) to a domain \(Dom(x_j)\). For each value \(v \in Dom(x_j)\), (i) \(A_j(v) = 0\) or \(f \s.t \ f \in F_j\) and (ii) \(\forall f \in F_j \exists \varepsilon \in Dom(x_j) \ s.t \ A_j(v) = f\).

Given these definitions, the TDG problem for selections with range predicates is equivalent to finding assignments \(A_1, \ldots, A_d\) such that the following system of equations is satisfied

\[
\forall_{1 \leq j \leq d} \forall_{1 \leq b(j)} \sum_{v \in B(i, x_j)} A_j(v) = |B(i, x_j)| \tag{7.1}
\]

\[
\forall_{1 \leq j \leq d} \forall_{1 \leq b(j)} \sum_{R \in cell(B(i, x_j))} |R| = |B(i, x_j)| \tag{7.2}
\]

\[
\forall_{1 \leq i \leq k} \sum_{R \in cell(Q_i)} |R| = |Q_i| \tag{7.3}
\]

\[
\sum_{R \in cell(A)} |R| = |T| \tag{7.4}
\]

\[
\forall_{1 \leq i \leq k} \forall_{R \in cell(Q_i)} |R| = |R_l| \tag{7.5}
\]

\[
\forall_{R_l} |R_l| \geq 0 \tag{7.6}
\]

\[
\forall_{R} |R| \geq 0 \tag{7.7}
\]

Equation 7.1 defines the mapping of the assignments \(A_j\) along each attribute \(x_j\) to the cardinality of each bucket \(|B(i, x_j)|\). Equation 7.2 maps the bucket cardinalities \(|B|\) to cell cardinalities \(|R|\).
Equation 7.3 ensures that the cardinality constraints associated with each query $Q_i$ are satisfied, while equation 7.4 ensures that the cardinalities of all the cells of a relation $A$ sum to the size of the relation. Finally, Equation 7.5 serves to ensure that the cardinality of a cell $R$ of $A$ that passes through the selection predicates of a query $Q_i$ remains unchanged.

**Lemma 6.** The TDG problem for selection queries with distributional constraints on the selection columns over a single relation is NP-hard

**Proof.** Consider an instance of the subset-sum problem with the capacity $N$ and a set of frequencies $f_1, \ldots, f_m$ s.t $\sum_{1 \leq i \leq m} f_i = S$. Define two test queries $Q_1$ and $Q_2$ on a relation $A$ with constraints $|A| = S$, $Q_1: |\sigma_{x < 100}A| = N$ and $Q_2: |\sigma_{x \geq 100}A| = S - N$. Additionally specify a distributional constraint $F$ on attribute $x$ such that $F$ specifies the frequencies $\{f_1, \ldots, f_m\}$. This is an instance of the TDG problem on a single relation with a single range selection predicate.

The above discussion assumed that we only have range predicates. Equality predicates of the form $x_j = C$ can be dealt with accordingly, by defining special buckets which correspond to a single element $C$. The rest of the analysis remains unchanged. Also, note that in the absence of distributional constraints $F_j$, the problem reduces to a simple set of linear equations which can be solved in polynomial time.

We discuss our approximation scheme for the TDG problem with selections in Section 7.4.3.

### 7.3.2 Aggregation Operators

In the previous section, we demonstrated that the TDG problem for even simple selection queries is NP-hard. In this section, we consider the TDG problem for an aggregation operator $O$ which performs a group-by along a grouping column $x_g$ on the output of its child $O_r$. We assume that a distribution constraint $F_{g,i}$ is defined for each query $Q_i$. Note that in this case, there is a separate frequency distribution $F_{g,i}$ for each query $Q_i$.

As with selection operators, we define an assignment $A_g(Q_i, v)$ to be a mapping between the values $v \in \text{Dom}(x_g)$ and the frequencies defined by the distribution function $F_{g,i}$. The only difference from the previous definition of assignments (for selections) is that the assignment is per query i.e. $A_g$ has the query $Q_i$ as a parameter as well.

On similar lines, for each cell $R$ in the output of $O_r$ we define $A_g(R, v)$ as the frequency of value $v \in \text{Dom}(x_g)$ in the cell $R$.

Then the TDG problem for aggregation operators requires defining an assignment $A_g$ satisfying
7.3. Analysis

Each of the frequency distributions \( F_{g,1}, \ldots, F_{g,k} \) such that:

\[
\forall 1 \leq i \leq k \forall v \in \text{Dom}(x_g) \sum_{R \in \text{cell}(Q_i(Q_r))} A_g(R, v) = A_g(Q_i, v) \tag{7.8}
\]

\[
\forall 1 \leq i \leq k \sum_{v \in \text{Dom}(x_g)} A_g(R, v) = |R| \tag{7.9}
\]

\[
\forall 1 \leq i \leq k \sum_{R \in \text{cell}(Q_i(Q_r))} \sum_{v \in \text{Dom}(x_g)} A_g(R, v) = |Q_i(Q_r)| \tag{7.10}
\]

\[
\forall R \ |R| \geq 0 \tag{7.11}
\]

Equation 7.8 ensures that the assignments on cells \( A_g(R, v) \) and assignments on queries \( A_g(Q_i, v) \) are consistent. Equation 7.9 defines cell cardinalities \( |R| \) in terms of the sum of all the frequencies assigned to the cell \( R \). Equation 7.10 is identical to Equation 7.3 defined previously, and ensures that the cardinality constraints on the cell are satisfied.

**Lemma 7.** The TDG problem for aggregation queries with distributional constraints on the grouping column is NP-hard

*Proof.* Consider the following problem, which we term as Exclusive Packing \( \text{EXCL} - \text{PACK} \):

We are given \( k - 1 \) disjoint sets of objects, \( S_1, \ldots, S_{k-1} \). Each object \( o \) has a weight \( w(o) \). Let the function \( s(o) \) return the set to which \( o \) belongs. We are also given \( b \) bags \( B_1, \ldots, B_b \), with capacities \( C_1, \ldots, C_b \) respectively.

Finally, we know that

\[
\sum_{i=1}^{b} C_i = \sum_{i=1}^{n} \sum_{o \in S_i} w(o)
\]

That is, the sum of the weights of all objects = sum of capacities of bags. The goal is to assign objects \( o \) to bags \( B \) such that

\[
\forall_{i=1}^{b} \forall_{o_x, o_y \in B_i} s(o_x) \neq s(o_y)
\]

(i.e no two objects of the same set are assigned to the same bag)

such that

\[
\sum_{o \in B_i} w(o) = C_i
\]

Now \( \text{EXCL} - \text{PACK} \) can be reduced to the TDG problem for groups as follows. Queries \( Q_1, \ldots, Q_{k-1} \) correspond to sets \( S_1, \ldots, S_{k-1} \). The queries are disjoint i.e \( \forall_{i,j} Q_i \cap Q_j = \phi \). Each query \( Q_i \) is associated with a frequency distribution on a grouping column \( x_g \) as \( F_{g,i} \) defined by the weights of objects \( \{ w(o) : s(o) = S_i \} \). Finally, query \( Q_k = \bigcup_{1 \leq i \leq k-1} Q_i \), with frequency distribution \( F_{g,k} \) on the grouping column defined by the bag capacities \( C_1, \ldots, C_b \).

Essentially, the bags act as the group-frequency distribution for the query \( Q_k \) that is a union of all the other queries. The constraint that no two objects of the same set are assigned to the same bag ensures that each group frequency gets assigned a different value.

We now show that \( \text{EXCL} - \text{PACK} \) is NP complete. The proof proceeds as a reduction from the following problem, referred to as the 3 dimensional matching (3-DM).
3DM: Given set $M \subseteq X \times Y \times Z$ where $X$, $Y$ and $Z$ are disjoint sets having the same number $q$ of elements. Does $M$ contain a matching $M' \subseteq M$ such that $|M'| = q$ and no two elements of $M'$ agree on any coordinate? i.e each element of the sets $X, Y, Z$ is present in only one triple $(x, y, z) \in M'$.

The reduction essentially mimics the first part of the proof that 3-Partition is NP complete in the Garey and Johnson textbook [46].

Let $X = \{x_1, \ldots, x_q\}$, $Y = \{y_1, \ldots, y_q\}$ and $Z = \{z_1, \ldots, z_q\}$, and $M \subseteq X \times Y \times Z$ denote an arbitrary instance of 3DM. We may assume without loss of generality that $|M| \geq q$.

We generate a corresponding instance of EXCL - PACK with $n = 4$, and $b = |M|$ bags, each having the same capacity. Each triple $m_{ijk} = (x_i, y_j, z_k) \in M$ contributes exactly one element each to 4 sets $S_x, S_y, S_z, S_m$.

For any element $t \in X \cup Y \cup Z$, let $N(t)$ denote the number of triples $\in M$ that contain $t$. There element $t$ will therefore create $N(t)$ elements $t[1], \ldots, t[N(t)]$ in either $S_x$ or $S_y$ or $S_z$. We denote one of these elements ($t[1]$) as the corresponding to the “matched” entry, and the other $N(t) - 1$ entries $t[2], \ldots, t[N(t)]$ as the “unmatched” entries.

Finally, let $r = 32q$.

We set weights to elements as per the following formulas

$$w(x_i[1]) = 10r^4 + ir + 1$$
$$\quad 1 \leq i \leq q$$
$$w(x_i[l]) = 11r^4 + ir + 1$$
$$\quad 1 \leq i \leq q, \quad 2 \leq l \leq N(x_i)$$

$$w(y_j[1]) = 10r^4 + jr^2 + 2$$
$$\quad 1 \leq j \leq q$$
$$w(y_j[l]) = 11r^4 + jr^2 + 2$$
$$\quad 1 \leq j \leq q, \quad 2 \leq l \leq N(y_j)$$

$$w(z_k[1]) = 10r^4 + kr^3 + 4$$
$$\quad 1 \leq k \leq q$$
$$w(z_k[l]) = 8r^4 + kr^3 + 4$$
$$\quad 1 \leq k \leq q, \quad 2 \leq l \leq N(z_k)$$

This takes care of assigning weights to elements in the sets $S_x, S_y, S_z$. Additionally there is another set $S_m$ corresponding to the triples $m_{ijk} \in M$ where $m_{ijk} = (x_i, y_j, z_k)$. We denote the element corresponding to $m_{ijk}$ by $u_{ijk}$ and set its size as

$$w(u_{ijk}) = 10r^4 - kr^3 - jr^2 - ir + 8$$

Finally, we define $|M|$ bags each with capacity $C = 40r^4 + 15$. 

Observe that the only way to satisfy the capacity of a bag is to either pack \((u_{ijk}, x_i[1], y_j[1], z_k[1])\) into a bag, or to pack \((u_{ijk}, x_i[l^1], y_j[l^2], z_k[l^3])\) where \(l^1, l^2, l^3 \geq 2\). The first case corresponds to selecting the \(m_{ijk}\) in the matching \(M'\), while the second case corresponds to rejecting \(m_{ijk}\) from the matching i.e \(m_{ijk} \in M - M'\).

We now show that there is a solution to the decision version of the exclusive packing problem designed above iff there exists a matching \(M' \subseteq M\).

First suppose there does exist a matching \(M' \subseteq M\). For each \(m_{ijk} \in M'\) pack \(u_{ijk}, x_i[1], y_j[1], z_k[1]\) in a bag \(B\). For each \(m_{ijk} \in M - M'\) pack \((u_{ijk}, x_i[l^1], y_j[l^2], z_k[l^3])\) in a bag, where \(l^1, l^2, l^3 \geq 2\). We are guaranteed enough unmatched elements to perform this packing due to our construction above.

Now to see the reverse case, i.e if there is a valid exclusive packing, then there is a matching.

Now suppose there is a valid exclusive packing. Suppose bag \(B\) is packed with elements \(u, x, y, z\).

Now as illustrated above, \(w(u) + w(x) + w(y) + w(z) = C = 40r^4 + 15\) iff \(u, x, y, z\) correspond to either \((u_{ijk}, x_i[1], y_j[1], z_k[1])\) or \((u_{ijk}, x_i[l^1], y_j[l^2], z_k[l^3])\) where \(l^1, l^2, l^3 \geq 2\). This can be easily decided by observing the weights of the elements \(x, y, z\).

Now if \((u, x, y, z)\) is of the form \((u_{ijk}, x_i[1], y_j[1], z_k[1])\) then add \(m_{ijk}\) to the matching \(M'\). Otherwise \(m_{ijk} \in M - M'\). It is easy to see that \(|M'| = q\) and \(M'\) is a valid 3 dimensional matching.

Therefore the EXCL – PACK problem is NP complete. As a consequence, the TDG problem for aggregation operators is NP hard.

\(\square\)

### 7.3.3 Join Operators

Consider an instance of the TDG problem for a join operator \(O\), in which each test case \(Q_i(O) = Q_i(O_l) \forall x_{i}=x_{r}\) \(Q_i(O_r)\) is defined by two join inputs \(O_r\) and \(O_l\), with a distributional constraint \(F_{r,i}\) on the join attribute \(x_r\) of the input \(Q_i(O_r)\), and a key constraint \(F_{l,i}\) on \(x_l\) at the other input \(Q_i(O_l)\). As with aggregations, we define two assignments \(A_l(Q_i, v)\) and \(A_r(Q_i, v)\). \(A_l\) corresponds to the key distribution while \(A_r\) corresponds to the non-key distribution.

Note that satisfying each distribution constraint \(F_{r,i}\) on the input \(Q_i(O_r)\) simultaneously over the set of test cases \(Q_1, \ldots, Q_k\) requires solving equations 7.8 to 7.11 as listed for aggregations. The only difference is that \(A_r\) replaces \(A_g\) in the equations. This is an NP-hard problem, as stated in Lemma 7. Consequently, the TDG problem for joins is an NP-hard problem as well.
\textbf{Chapter 7. TDGen: Targeted Database Generation}

In addition, the following other equations must be satisfied as well

\begin{align}
\forall v \in \text{Dom}(x_j) \quad & \exists R_l, R_r \neq R_l^1 \{ A_l(R_l^1, v) = A_l(R_l^2, v) = 1 \} \\
\forall 1 \leq i \leq k \forall v \in \text{Dom}(x_j) \quad & \sum_{R_l \in \text{cell}(Q_i(O_i))} A_l(R_l, v) = A_l(Q_i, v) \\
\forall 1 \leq i \leq k \forall v \in \text{Dom}(x_j) \quad & \sum_{R_l \in \text{cell}(Q_i(O_i))} A_l(R_l, v) = |R_l| \\
\forall |R_l| \geq 0 \\
\forall R = R_l \bowtie R_r \quad & \sum_{v \in \text{Dom}(x_j)} A_l(R_l, v) \times A_r(R_r, v) = |R| 
\end{align}

Equations 7.12 and 7.13 ensure that the distribution on the left input is a key distribution. Equations 7.14 and 7.15 ensure that the cell cardinalities are greater than 0, and are consistent with the assignments. Finally, Equation 7.16 ensures that the output cell cardinality is consistent with the distribution of the join keys. This represents the simple case when the join is the topmost join of the plan. Further complications arise due to additional constraints on the output data, as detailed in the next section.

\section{The TDGen Framework}

In the previous section, we have shown that the TDG problem is NP-hard even for simple queries involving only selection, join or aggregation operators. This is due to distributional constraints on the attributes, and the inter-dependency between the test cases. In a general setting, we can expect test cases to consist of queries involving all three kinds of operators, with multiple distributional constraints on relevant attributes. To handle these settings, we have developed the \textit{TDGen} framework for targeted database generation. \textit{TDGen} provides best-effort solutions for satisfying the distributional constraints, while ensuring that the cardinality constraints are satisfied.

\subsection{Overview}

\textit{TDGen} takes as input a testing plan consisting of multiple test cases, as defined in Section 7.2. It constructs a logical operator tree for the testing plan, and operates on it. For example, Figure 7.1(a) is the logical operator tree for the test cases described in Figures 7.1 (b) and (c). Given this common tree, \textit{TDGen} operates in two phases:

\textbf{Phase 1: Bottom-up Constraint Propagation.} As described in Section 7.3, the TDG problem is NP-hard, even without considering the interdependencies between operators in the logical operator tree. The first phase of \textit{TDGen} captures the dependencies by propagating constraints from the leaves of the logical tree up to the root i.e in a bottom-up fashion. Specifically, \textit{TDGen} focuses on propagating two types of constraints: cardinality constraints and distinct value constraints. Cardinality constraints specify that the total number of tuples in a set of cells cannot...
exceed a certain threshold. Distinct value constraints specify a bound on the number of distinct values along a group of attributes in a set of cells. Both these constraints arise as a consequence of our requirement that joins have at least one side with a key constraint. Cardinality constraints propagate from the non-key side of the join, while distinct value constraints arise due to the key side. We describe the constraint propagation phase of TDGen in Section 7.4.2.

**Phase 2: Top-down Data Generation.** The second phase of TDGen operates in a top-down fashion, from the root of the tree to the leaves. For each operator $O$, a data instantiation module is called that takes as input a multidimensional histogram representing the output of $O$ along the attributes $O.parAtts$ for each of the test queries $Q_1,\ldots,Q_k$. Given this, the module instantiates data for the relevant attributes of $O$ i.e $O.relAtts$, and combines it with the previously generated output data to produce multidimensional histograms representing the input of $O$ i.e $O_r$ along $O_r.parAtts$ (and if $O$ is a join, also $O_l.parAtts$). This process is repeated until we reach the scan operators of the tree, where the multidimensional histogram is used to generate data for the base tables. We provide further details of the data generation framework in Section 7.4.3.

### 7.4.2 Constraint Propagation

We consider two forms of constraint propagation from the leaves to the roots of the operator tree: cardinality constraints, and distinct value constraints. We describe our techniques for propagating such constraints in this section.

**Propagating Cardinality Constraints**

The goal of the process of propagating cardinality constraints is to assign a cardinality $|R|$ to each cell $R$ at each intermediate operator $O$ of the query. The process proceeds in two steps.

**Initialization:** The initialization step occurs at each table scan operator $O$, and its parent selection operator $O_p$ (if present). Let $A$ be the table scanned by $O$ i.e $\{A\} = O.tables$, and let its cardinality be $|A|$. Then, the initialization step involves solving the equations 7.3 to 7.7 from Section 7.3.1. This is the subset of the equations described in Section 7.3.1, which do not take into account the assignment of frequencies to cells. They can be easily solved using a linear equation solver. The initialization process described here is encapsulated in the call $O.initialize()$ in Algorithm 15.

**Propagation:** At the end of the initialization step, we have cell cardinalities $|R|$ associated with each cell of each base table, and at the output of each selection operator. The goal of the propagation phase is to propagate these cell cardinalities up the join tree, and assign cell cardinalities to the join output. The propagation process is described in Algorithm 15. For each join operator $O$, the function $GenCard$ is invoked. $GenCard$ generates the set of equations corresponding to the output of the join (lines 9 - 14). After that, $GenCard$ generates additional equations due to the constraints
Propagating Distinct Value Constraints

Distinct value (DV) constraints arise in the TDG problem due to the key constraint on the join attribute of the left input $O_l$ of a join operator $O$. As a consequence, for each cell $R_l$ of the left input such that $|R_l| = 0$, $GenCard$ ensures that the parents of $R_l$ cannot contain any tuples (lines 18-22). Finally, $GenCard$ solves the generated equations to generate cell cardinalities $|R_l|$ for each cell in its output.
Algorithm 16 Propagation of DV Constraints

```plaintext

Struct DVBound
    int Limit;
    List Attrs; //List of Attributes
    Set CellSets; //A set of sets of cells

DVBound(int Limit, List Attrs, Set CellSets) //Constructor

1: Function PropDV(Operator O)
2:    O.DVs = {}
3:    for all R_l ∈ cell(O_l) do
4:        newdv = new DVBound([R_l],O_l.parAtts,parents(R_l))
5:            O.DVs.add(newdv)
6:    end for
7:    for all dv ∈ O_l.DVs do
9:            O.DVs.add(newdv)
10:   end for
11:   for all dv ∈ O_r.DVs do
12:       if O_r.relAtts ∈ dv.Attrs then
14:       else if dv.Attrs ∩ O_r.parAtts ≠ ∅ then
16:       end if
17:           O.DVs.add(newdv)
18:   end for

19: Function GenParents(Set CellSets)
20:    ParSets = {}
21:    for all Set S ∈ CellSets do
22:        PS = \bigcup_{R ∈ cell(S)} parents(R)
23:        ParSets.add(PS).
24:    end for
25:    return ParSets

26: Function SplitLeft(Set CellSets)
27:    ParSets = {}
28:    for all Set S ∈ CellSets do
29:        for all Cell R_l ∈ O_l do
30:           PS = \bigcup_{R ∈ cell(S)} (parents(R) ∩ parents(R_l))
31:           ParSets.add(PS).
32:        end for
33:    end for
34:    return ParSets
```

on the left input is propagated (lines 7-10) with an invocation of GenParents (lines 19-25). Distinct value constraints from the right input that do not include the join attribute in a similar fashion as well (lines 14-15).

However, if the distinct value constraint on the right input is specified on the join attribute, propagation proceeds differently. In this case (lines 12-13), the output DVBound is defined on a
set comprising multiple sets, each generated by a different cell \( R_l \) of the left input \( O_l \) (function \( \text{SplitLeft} \) lines 26-34). This is to take into account the fact that any two tuples of the left input must differ on the join key. In addition, the new \( DVBound \) is defined on the union of attributes from the left input, and right \( DVBound \).

**Example 21.** Consider a join operator with join condition \( A.x = B.y \), with a key constraint on \( A.x \). \( A \) is the left input with attributes \((A.x, A.w)\). Suppose we have a distinct value constraint of 5000 tuples on attributes \((B.y, B.z)\) on a cell \( R_r \) of the right input. Let \( R_r \) join with cells \( R_{1l}^1, R_{1l}^2 \) of the left input, with cardinality 3000, and 8000 tuples respectively. Since no value of \( A.x \) can be duplicated in \( R_{1l}^1 \) and \( R_{1l}^2 \), we require that the sum of the number of distinct values of \((B.z, A.w)\) in the output cells \( R_r \Join R_{1l}^1, R_r \Join R_{1l}^2 \) be less than 5000.

One final thing to note is that at each propagation step, we reduce the Limit on distinct values by the number of distinct sets in the CellSets set. This is because some tuples from the right and left inputs may not join, and need to be considered while propagating the \( DVBound \). In principle, this may cause some \( DVBounds \) to go below zero, making data generation impossible. This case is however, unlikely to happen for queries on sufficiently large tables, since the size of CellSets is typically very small.

In the next section, we describe our data generation algorithms, which take into account the fact that the distinct value bound must always be preserved.

### 7.4.3 Data Generation

Once the constraints are propagated to the top of the logical operator tree, \( \text{TDGen} \) proceeds to generate data in a top-down fashion, as illustrated in Figure 7.3. Data generation starts at the root of the tree, which instantiates its input data and stores it in a temporary file. Each subsequent operator takes as input the data generated by its parent, and generates child data accordingly. This process continues until the Selection/Scan operators, which generate the final base tables as flat files in a CSV format. Before describing the individual data generation mechanisms for various operators, we outline some common modules required by our algorithms:

**Distribution Generator:** Given a specification of a data distribution \( F \), a distribution generator generates the frequencies in \( F \) in a non-increasing order. A distribution generator supports two functions, \( \text{GetNext} \), and \( \text{Adjust} \). \( \text{GetNext} \) returns the next frequency in \( F \) in non-increasing order. \( \text{Adjust} \), adjusts the frequencies of \( F \) to handle errors in data generation due to generation of fewer or more tuples than specified by the previously returned frequency. In this work, we adapt the distribution generators presented in [51], however our techniques are amenable to any distribution generator that satisfies the properties outlined above.

**Unique Value Generator:** A unique value generator always returns a unique value between a lowerbound and an upperbound each time a \( \text{GetNextUnique} \) function is invoked.
7.4. The TDGen Framework

Each operator takes as input a representation of the data generated by its parent operator. We store this data as multidimensional histograms associated with each cell of the parent. An operator initializes Distribution Generators and Unique Value Generators for each relevant attribute to be generated by it. Data generation proceeds in rounds, with the operator attempting to generate tuples satisfying the distribution constraints in a best-effort fashion, while also satisfying the output data generated by the parent, cardinality constraints and associated distinct value constraints.

We present the algorithms associated with operators in a top-down fashion, first describing aggregation operators; building upon the framework for aggregations to handle join operators; and finally describing the techniques for Selection operators and base table scans.

Aggregations

In the model of testing plans considered in this work, aggregation (group-by) operators can occur only at the root of the logical operator tree. As stated in Section 7.3.2, the TDG problem for aggregation operators with both cardinality and distributional constraints is NP hard. Moreover, the $k$-dimensional matching problem, which reduces to the TDG problem for aggregations is known to be difficult to approximate [66, 71]. As a consequence, in this section, we utilize a lightweight greedy approximation algorithm, which works well in practice.

Algorithm 17 describes the greedy approximation algorithm utilized for generating aggregation
Algorithm 17 Packing groups

1: Function PackGroups(O, x_g)
2: while packed == false do
3:  \forall Q_i : f_i = GetNext(F_{g,i})
4:  if \exists f_i : f_i > 0 then
5:    packed = true; break;
6: end if
7:  Group g = GetNextUnique(x_g)
8:  \forall cell R : SetCaps(R)
9: while \exists R s.t. R.cap > 0 do
10:    Set NonEmpty = \{ R : R.cap > 0 \}
11:    R_m = maxcap(maxoverlap(NonEmpty))
12:    Assign(R_m, R_m.cap, g)
13:    UpdateDV(R_m)
14:    for all Q_i s.t R_m \in Q_i do
15:        f_i = f_i - R_m.cap
16:    end for
17:    \forall cell R \in NonEmpty : SetCaps(R)
18: end while
19: \forall Q_i Adjust(F_{g,i}, f_i)
20: end while

21: Function SetCaps(cell R)
22: R.overlap = |Q_i : R \in cell(Q_i)|
23: R.cap = |R| - R.curr
24: if checkDV(R) then
25:    R.cap = min(R.cap, \{ f_i : R \in Q_i \})
26: end if

data. The algorithm proceeds in rounds. In each round, the distribution generators (line 3) and unique value generators (line 7) are invoked. Given, the set of group frequencies returned by distribution generator, the main loop of the procedure (lines 9-18) makes a best-effort attempt to assign the frequencies to the cells. In each iteration of the loop, the capacity and overlap of each cell is computed through an invocation of the SetCaps procedure (lines 21-26). The capacity refers to the maximum frequency group that can be assigned to the cell. The overlap here refers to number of queries containing the cell. In each iteration, we assign the group to the cell which has the maximum overlap, breaking ties by capacity (line 11). This is the core of the group assignment heuristic, which can be changed if required. This process repeats until no cell has a non zero capacity.

Algorithm 17 takes into account distinct value constraints in lines 24-26. If a cell can take only one more distinct group, the SetCaps procedure ensures that the cell is assigned a group with frequency |R| - R.curr, which makes up for the gap. If DV constraint is triggered, the procedure effectively ignores the currently generated frequency, assigning the cell its full capacity at one go.

Algorithm 17 is a single pass algorithm that guarantees satisfying the cardinality constraints. The best-effort nature of the distributional constraints arises due to the distinct value constraints,
and the potential incompatibility of the group distributions of the multiple test cases under consideration.

Joints

We now describe our procedure for generating the input data at join operators. Join operators can be either at the root or at an intermediate node of the logical operator tree. Consequently, our procedures may need to take into account the data distribution at the output node of the plan. Moreover, in the class of testing plans considered in this work, one of the inputs (the left input) has a key constraint on the join attribute. As a consequence, any tuple of the right input may join with at most one tuple of the left input, while any tuple of the left input can join with multiple tuples of the right input. In addition, we note that the process of distributing the frequencies of the join attribute among the cells of the right (non-key) input is equivalent to the group assignment problem discussed in the previous section, and our techniques build upon those introduced there.

Algorithm 18 describes our procedure for assigning values to frequencies along the join attribute. As a first step (lines 3-15) function PackJoin checks if there is a parent operator to the join node, and if so, whether the data generated by it contains any attributes from the left input \( O_l \) of the join. If so, PackJoin groups the parent data on the left input columns, and orders it by the frequency (line 5). This operation is performed for the set of parent cells for each left cell \( R_l \). We note that this operation involves sorting twice: once for the grouping, and once for the order-by clause. This is effectively the most expensive operation in the entire TDGen framework, and is performed only if required. PackJoin stores the data generated by the parent in a disk based priority queue (lines 5-6). In the alternate case, when no parent or left attributes are present, PackJoin simply maintains a priority queue ordered by the sum of parent cell cardinalities for each left input cell \( R_l \) (lines 10-15).

Once the preprocessing phase is completed, PackJoin proceeds in rounds like the PackGroups function from Algorithm 17. In each round, the distribution generators (line 19) and unique value generators (line 23) are invoked to generate a distribution of frequencies for the right input, and a unique value for the join attribute respectively. Given these frequencies, they can either be assigned to a value that they does not join with any tuple from the left input, or they can be assigned to a value that joins with a tuple in cell \( R_l \). This choice is made greedily, depending on whether the topmost entry in the priority queue corresponding to the output of the left cells exceeds the flexibility of the right input (i.e the number of tuples that may not join) (lines 24-28).

After the decision on whether or not to join is made, the main loop of the procedure (lines 30-39) is almost identical in functionality to the corresponding loop in the PackGroups procedure (Algorithm 17). It also involves invocations of a SetCaps procedure (lines 45-59) which is similar in functionality. One major difference is that data generation for join operators requires checking distinct value constraints at both the left and right input cells, ensuring that the constraints are not
Algorithm 18 Packing Joins

1: Function PackJoin(O, x_r, x_l)
2:   leftAtts = O.parAtts \cap O_l.parAtts
3: if \exists O_p \land leftAtts \neq \phi then
4:   for all R_l \in cell(O_l) do
5:     PriorityQueue R_l.PQ = parents(R_l).data Group by leftAtts Order by count(*)
6:     PriorityQueue GlobalPQ.add(R_l.PQ.peek())
7:   end for
8: hasleft = true
9: else
10: for all R_l \in cell(O_l) do
11:   R_l.output = \sum R_r \in parents(R_l) \mid R_r \mid
12:   PriorityQueue GlobalPQ.add(R_l.PQ, R_l.output)
13: end for
14: hasleft = false
15: end if
16: \forall R_r \in cell(O_r) : R_r.flex = \mid R_r \mid - \sum R_r \in parents(R_r) \mid R_r \mid
17: flex = \sum R_r \in cell(O_r) R_r.flex
18: while packed == false do
19:   \forall Q_i : f_i = GetNext(F_r,i)
20: if \exists f_i : f_i > 0 then
21:   packed = true; break;
22: end if
23: JoinKey v = GetNextUnique(x_r, x_l)
24: if flex > GlobalPQ.peek() then
25:   doesjoin = false; Cell R_l = NULL
26: else
27:   doesjoin = true; Cell R_l = GlobalPQ.poll()
28: end if
29: \forall R_r \in cell(O_r) : SetCaps(R_r, R_l, doesjoin)
30: while \exists R_r s.t. R_r.cap > 0 do
31:   Set NonEmpty = \{ R_r : R_r.cap > 0 \}
32:   R_m = maxcap(maxcard(NonEmpty))
33:   GenerateRightTuples(R_m, R_m.cap, v)
34:   UpdateDV(R_m); UpdateDV(R_l)
35:   for all Q_i s.t R_m \in Q_i do
36:     f_i = f_i - R_m.cap
37:   end for
38:   \forall R_r \in NonEmpty : SetCaps(R_r, R_l, doesjoin)
39: end while
40: if doesjoin then
41:   GenerateLeftTuple(R_l, 1, v)
42: end if
43: \forall Q_i : Adjust(F_r,i, f_i) // Also adjust diff and the priority queues
44: end while

45: Function SetCaps(R_r, R_l, doesjoin)
46: R_r.card = \mid Q_i : R \in cell(Q_i) \mid
47: if doesjoin \land \mid R_r \otimes R_l \mid > 0 then
48:   if hasleft then
49:     R_r.cap = R_l.PQ.peek().project(R_l \otimes R_r)
50: else
51:     R_o = R_l \otimes R_r
52:     R_r.cap = \mid R_o \mid - R_o.curr
53: end if
54: else
55: R_r.cap = R_r.flex
violated at either side. In each case, \textit{SetCaps} effectively ignores the currently generated frequency and proceeds to ensure that the distinct value bound is not violated.

**Selections**

**Algorithm 19** Packing Selection Attributes

```
1: Function PackSel(O, R)
2: for all \( x_s \in O\.relAtts \) do
3: \hspace{0.5em} PackSelAtt(O, \( x_s \))
4: end for
5: for all Cell \( c \in cell(R) \) do
6: \hspace{0.5em} GenerateData(c)
7: end for
8: Function PackSelAtt(O, \( x_s \))
9: for all Buckets \( B \in buckets(x_s) \) do
10: \hspace{0.5em} \( B\.t = 0 \)
11: end for
12: while \( \{ f = \text{GetNext}(Fs) \} \neq 0 \) do
13: \hspace{0.5em} Bucket \( B_m = \min_{B \in \text{buckets}(x_s)}(B\.t + \frac{f}{|B|}) \)
14: \hspace{0.5em} Value \( v = \text{GetNextUnique}(x_s, B_m) \)
15: \hspace{0.5em} if \( B_m\.curr + f > |B_m| \) then
16: \hspace{1.5em} overflow = \( B_m\.curr + f - |B_m| \)
17: \hspace{1.5em} \( f = |B_m| - B_m\.curr \)
18: \hspace{0.5em} end if
19: \hspace{0.5em} Assign(f, v)
20: \hspace{0.5em} \( B_m\.t = B_m\.t + \frac{f}{|B_m|}; B_m\.curr = B_m\.curr + f \)
21: \hspace{0.5em} if overflow then
22: \hspace{1.5em} Adjust(F_s, overflow)
23: \hspace{0.5em} end if
24: end while
```

Selection operators, paired with associated table scans form the leaf nodes of the logical operator tree. In the context of this work, we consider selection operators defined on attributes that do not participate in further join or aggregation operations. Moreover, unlike joins, selection operators produce at most one tuple of output for each tuple consumed. Consequently, our data generation procedures do not need to consider the distribution of data at the output of the operators. Algorithm 19 describes our procedures for generating data satisfying the constraints at selection operators. Procedure \textit{PackSel} iterates over the selection attributes, invoking function \textit{PackSelAtt} for each attribute (lines 2-4). \textit{PackSelAtt} attempts to pack the frequencies generated by the distribution generator (line 12) in the buckets defined on the domain of the selection attribute.

In Section 7.3.1, we demonstrated the NP hardness of this problem through a reduction from the Subset-Sum problem. A related problem is the problem of minimum makespan scheduling on uniform processors [68]. Essentially, bucket sizes can be mapped to processor speeds, and frequencies can be mapped job sizes. Although Hochbaum and Shmoys [68] provide a PTAS for
the scheduling problem, it is not suitable for practical implementation in our setting. Instead we adopt the greedy LPT (Largest Processing Time) approximation scheme [48] which is known to be 2-optimal.

Procedure PackSelAtt in Algorithm 19 describes the value-frequency assignment procedure for selection attributes. Each bucket \( B \) is associated with a cardinality \(|B|\), and a completion fraction \( B.t \) initialized to zero. In each iteration, PackSelAtt invokes the data generator (line 12) to obtain a frequency \( f \). The frequency \( f \) is assigned to the bucket with minimum value of \( B.t + \frac{f}{|B|} \) (line 13-14). This procedure is repeated until all frequencies are assigned. In order to preserve the bucket cardinalities, overflows are dealt with by adjusting the data generator (lines 15-18;21-23). The assigned frequency distributions are stored in a temp file for the next step of data generation.

PackSelAtt is invoked for each selection attribute of the table associated with the operator. Once this procedure is completed, data generation proceeds in a cell-wise manner, combining the data instantiated by the parent operator (if any), and the bucket-wise data distribution generated by the procedure. This is encapsulated in the call to GenerateData, pseudocode for which is not provided for conciseness.

### 7.5 Evaluation

In this section, we describe the design and evaluation of the TDGen prototype. The focus of our evaluation is primarily on data generation times (Section 7.5.2), but we also provide accuracy results in Section 7.5.3. Before describing our results, we first present our system design and experiment setup.

#### 7.5.1 System Design and Setup

The TDGen system was implemented in Java in approximately 7.5K lines of code. Figure 7.4
7.5. Evaluation

describes the overall design of the TDGen system. TDGen takes as input a specification of the testing plan, which describes the table sizes and schema, the query template, and individual test cases annotated with cardinality and distributional constraints. The parser converts this specification into a join tree with each node partitioned into a set of cells. This join tree is then utilized by the Constraint Propagation module, which implements the cardinality and distinct value propagation techniques described in Section 7.4.2. This module utilizes the GNU Linear Programming Kit [1] as a linear equation solver. The annotated join tree is then utilized by the data generation module (described in Section 7.4.3) to generate the data in a top-down fashion, with the intermediate and base tables written to disk. TDGen generates base tables as Comma-Separated Values (CSV) files, which can then be loaded in any database system.

Our evaluation of TDGen was conducted on a machine running Suse Linux with 4GB memory and 3.60GHz clock speed with a single SATA disk drive of size 250 GB. The disk was measured to have streaming read/write speed of 50 MB/s.

7.5.2 Data Generation Times

The objective of this section is to measure the execution times of TDGen required for generated large databases. Before we describe our results, we present for the purpose of comparison, a brief overview of previously reported data generation times in literature. We note however, that these systems solve somewhat different problems, making a direct apples to apples comparison infeasible.

The DGL system for flexible database generation [20] is designed to process database specifications written in a special purpose Data Generation Language (DGL). While DGL is designed to be flexible, it is not query-aware, and therefore cannot guarantee that a test query satisfies certain properties. In the DGL paper [20] results are reported for 1 GB synthetic databases having 4.5 - 50 million rows, with data generation times between 4 - 13 minutes. The variation in data generation times is a result of the variation in the complexity of the data specification.

The QAGen system [15] is designed for query-aware database generation. QAGen generates a test database corresponding to a single test query specification. The functionality provided by QAGen and TDGen significantly overlaps. QAGen supports a somewhat broader class of query plans; however TDGen supports multiple test queries simultaneously, and a more general class of distributional constraints. The QAGen paper [15] reports data generation times for generating a relevant subset of tables of the 1 GB TPCH specification with execution times varying from 256 - 1631 minutes, depending on the complexity of the test query. Note that QAGen is almost two orders of magnitude slower than DGL for generating test databases of comparable size.

In this section we demonstrate that TDGen can generate test data in time comparable to the the best published results on flexible database generation: DGL [20], while satisfying multiple test cases, and providing similar functionality to previous work on query-aware database generation: QAGen [15].
Single Table Queries

We present our results for test cases involving only a single table in Figure 7.5. We first describe our results for the scalability of TDGen with increasing table size, and then proceed to explore the scalability of TDGen with increasing number of test queries in the testing plan.

![Figure 7.5: Single Table: Varying Data Size](image)

**Varying Data Size:** In Figure 7.5, the plots marked \( (2, 3, 4 \text{ Sel Att}) \) describe data generation times for testing plans with three test queries, with the queries having two, three and four selection attributes respectively. Each test case specifies the selectivity of the associated query, and distributional constraints on each of the selection attributes. We vary the size of the table from 10 million to 100 million rows, while keeping the selectivity of each test case the same. The size of the CSV file produced varies from 132 MB to 1.4 GB for 2 selection attributes, 174 MB to 1.8 GB for 3 selection attributes, and 202 MB to 2.1 GB for 4 selection attributes.

The results plotted in Figure 7.5 demonstrate that TDGen scales linearly with increasing table size. For example, the data generation times for the 4 attribute case vary from 1’41” (for 10 million rows / 202 MB) to 17’23” (for 100 million rows / 2.1 GB). TDGen effectively has a write speed of approximately 2 MB/s in this experiment. While this time is significantly slower than the full streaming write speed of the disk (approximately 50 MB/s), this is due to the CPU processing accompanying the data generation. Moreover, as described in Section 7.4.3, satisfying selection constraints requires assigning the frequencies generated by the distribution generators to values, and storing the assignments in temp files. The final step processes all the temp files together to generate the appropriate data. As a consequence it involves multiple sequential reads in conjunction with a sequential write to the same disk.

Finally, we observe the plot marked as \( 3 \text{ Sel Att} + 1 \text{ Group Att} \) in Figure 7.5. In this plot, we have a distributional constraint on a grouping column at the output of the selection operator.
The table therefore has 3 selection attributes, and 1 grouping attribute. As in the other cases, the number of test queries was fixed at 3, with the table cardinality varying from 10 million to 100 million rows. Correspondingly, the CSV files varied from 210 MB to 2.2 GB in size. The data generation times scale linearly from 1’56” to 19’44”. The substitution of a selection column with a grouping column results in an approximately 15% increase in data generation times in comparison to the plot marked 4 Sel Att. This is a consequence of the introduction of a new operator in the query plan, and the overheads of passing data between the operators.

The experiments demonstrate the linear scalability of TDGen with increasing data size. We note that the data generation times described in this section are comparable to previously reported times for flexible database generation [20]. We next describe the effect of varying the number of queries in the testing plan.

![Figure 7.6: Single Table: Varying Number of Queries](image)

**Varying Number of Queries:** In the experiments described in Figure 7.6, we fix the table size to 50 million rows, while varying the number of queries from 2 to 10. As in Figure 7.5, we consider selection queries with two, three and four selection attributes, and a testing plan with three selection attributes, and a group-by operator on the output of the selection.

We first discuss the results on queries with only selections (marked as 2, 3, 4 Sel Att). Increasing the number of selection attributes increases the size of the data generated; however, increasing the number of test queries does not affect the size of the table. As Figure 7.6 demonstrates, the data generation times remain essentially unchanged with increasing number of queries in the testing plan. For example, for 4 selection attributes, the data generation times remain in a small range between 8’53” and 9’17”. This is because the number of test queries only affects the parsing and constraint propagation phases of TDGen. Compared to the cost of actual data generation, these phases are relatively inexpensive, and consequently the data generation times remain
unaffected. The small differences in the data generation times for the same number of selection attributes correspond to differences in the sizes of the CSV files produced.

We next explore the effect of increasing the number of test queries on a testing plan with both selections (on three attributes), and aggregations (marked as $3 \text{ Sel Att} + 1 \text{ Group Att}$ in Figure 7.6). Unlike the case of only selection attributes, the data generation times increase with with the number of test queries in the testing plan. This is because the data generation procedure for the aggregation operator generates only intermediate data produced at the output of the underlying selection operator. The size of the intermediate data increases with the number of test queries in the testing plan. Consequently, the work performed by the data generation procedure ($\text{PackGroups}$) also increases, and therefore the query generation times increase as well.

Together the two sets of results demonstrate that for single table queries, $TDGen$ scales linearly with data size, and is not significantly affected by the number of test queries in the testing plan. In the next section, we evaluate $TDGen$ for join queries.

Join Queries

In the previous section, we reported data generation times for queries that generate only a single table. In the following section, we consider testing plans with join queries that required $TDGen$ to generate two or more tables while satisfying multiple cardinality and distributional constraints. Figure 7.7 describes our results for queries involving one, two, and three join operators, with table sizes varying from 10 million to 100 million rows.

Figure 7.7(c) describes the templates for the testing plans considered in the experiments described in Figure 7.7(a) and (b). As stated previously, there is a key constraint on the left input to each join operator, while the right input has a distributional constraint (zipfian skew with different parameters). The two table query has the join condition $A \bowtie_i B$ and the three table query has the conditions $A \bowtie_i B \bowtie_j C$. Finally, we consider two variations of a four table query, with the final join condition as either (i) $A.k = D.k$ or (ii) $C.k = D.k$. We next explain the rationale behind considering these two cases.

In the discussion of our approach towards generating data for joins (in Section 7.4.3), we noted that the procedure adopted depends on whether the join output has attributes corresponding to the left (key) input of the join. If such attributes are present, the join output is to be grouped by the left attributes, and ordered by decreasing frequency. This process involves two external sorting steps, for the grouping and the ordering. Moreover, the output of the join is to be maintained as a priority queue, which can be expensive. If on the other hand, the join output does not contain any attributes from the left input, the join data generation step does not need to take into account the data distribution at the output of the operator, and is consequently much less expensive. Consequently, for the two and three table cases, and case (i) for four table joins, all the joins involve the rightmost input $A$ and therefore $TDGen$ does not invoke a sorting procedure. In
contrast, for case (ii) with the four table join, the output of the intermediate join operator contains an attribute \(C.k\) from the left input. Therefore, this case requires the intermediate data at the output of \(A \bowtie B \bowtie C\) to be sorted and maintained in a priority queue as discussed in Section 7.4.3.

We first consider the cases of two, three and four table joins that do not involve a sorting step. Figure 7.7(a) describes results for testing plans with three test cases, cardinality constraints on all intermediate tables, and distributional constraints on all relevant attributes at each operator. The table sizes are varied from 10 million rows each, to 100 million rows each. \(TDGen\) demonstrates a linear scaleup with increasing table size, with execution times varying from (for example) 12’13” to 126’51” for four table joins.

We next observe the special case (ii) for four table joins, where \(TDGen\) must perform a sorting step on the intermediate data at the output of \(A \bowtie B \bowtie C\). As with the previous case, we vary the
sizes of all four tables from 10 million tuples each to 100 million tuples each. The corresponding data generation times vary from 16’32” to 168’3”, demonstrating a linear scaleup with increasing table size. We note that in comparison to case (i), the data generation times are roughly a third higher. This is primarily due to the external sorting step, even though the data sizes produced are approximately the same.

Figure 7.7(b) describes the data generation times for the testing plans considered, plotted against total size of data generated. There are two observations in order here. Firstly, we note that for testing plans that do not require a sorting step, data generation times increase linearly with total size of data generated. This observation is independent of the number of operators in the testing plan, with data generation times of approximately 12 minutes per GB for two, three and four table joins. The second observation from Figure 7.7(b) is that even though the data generated for both cases of the four table testing plans is the same, TDGen with sorting (i.e case (ii)) generates data at approximately 16-17 minutes per GB. Although this is a third higher than TDGen without the sorting step, we note that such data generation speeds are comparable to the published results from the DGL system [20], while being significantly faster than QAGen [15]. One of the reasons why sorting is expensive is that our current implementation sorts java objects directly, and uses java serialization/deserialization methods to write/read the objects from the disk at the intermediate runs. These operations, as implemented in java, are fairly expensive though, and a reimplementation of the sorting and serialization functionality is likely to reduce the difference between the two cases even further.

7.5.3 Accuracy

![Figure 7.8: Accuracy: Selections](image)

Our focus in the evaluation has been primarily on the execution times of TDGen. In this section, we provide some results on the accuracy of the TDGen framework, with respect to how
closely it satisfies the test case specification. As noted in Section 7.4, TDGen is guaranteed to satisfy the cardinality constraints if the constraint propagation techniques detailed in Section 7.4.2 succeed. Specifically, we require that the equations corresponding to the cell cardinalities be satisfiable. This is easy to guarantee if the test cases do not conflict with each other. The other requirement is that the distinct value constraints resulting from the key constraint on the inner input of the join be propagated up the tree successfully. Note that at each propagation step in Algorithm 16, we reduce the limit on the number of distinct values permissible for the CellSet. This is because certain tuples in both inputs may not join. As a consequence, there is a small chance that the limit may fall below zero. However, this can happen only if the number of test cases is very large, and the data size is very small.

Since we are guaranteed satisfaction of the cardinality constraints, we instead consider how closely TDGen satisfies the distributional constraints. As stated previously, TDGen provides best-effort data generation for the distributional constraints. This is reflected in the design of the algorithms in Section 7.4.3 as well.

We consider a two table join template \((A \bowtie_j B)\) with selection predicates on \(A.x, B.y\) and \(B.z\). Relations \(A\) and \(B\) have 1 million rows each. The selection columns and corresponding distributional constraints are \((A.x, Z_1)\), \((B.y, Z_2)\) and \((B.z, Z_3)\). Figure 7.8 plots the 100 highest frequencies for each of these columns, and also plots the ideal frequencies as per the distribution alongside. As can be seen, the generated and ideal distributions are virtually indistinguishable, with only minor differences at a few points. In addition, the testing plan consists of two join queries, with the data distribution of the join column \(A.j\) specified as a key, and \(B.j\) specified as \(Z_{0.5}\) in each. Figure 7.9 plots the top 100 generated and ideal frequencies along \(B.j\) for both the test cases. In each case, the distribution remains unchanged.

In addition, for each of the illustrated plots, we fit an estimator for the exponent of the power
law distribution [37]. In each case, the exponent for the original, and the generated distribution was identical up to 3 decimal places. This example illustrates that the “best-effort” semantics of \textit{TDGen} work well in practice.

### 7.6 Conclusions

This work studies the Targeted Database Generation problem to satisfy multiple test cases simultaneously. In addition to proving various results on the hardness of various incarnations of the problem, we also demonstrate that practical solutions for targeted database generation are feasible. Our experimental evaluation demonstrates the \textit{TDGen} is significantly faster than previous efforts towards query-aware database generation, with data generation speeds close to the best known flexible database generation systems. This speedup is partly due to the focus on the class of testing plans considered in Section 7.2. A natural follow up question that we intend to explore is whether we can extend the scope of testing plans considered by \textit{TDGen}, without compromising on its practicality. Potential extensions include joins with neither input being a key, other relational operators, and removing the constraint on an attribute being relevant to at most one operator.
In Chapter 1 of this thesis, we stated that our objective was to explore the following class of problems.

**Problem Statement:** Given two of the following three: query $Q$, data $D$, and statistic $S$; and a template for the unspecified element (either $Q$, or $D$, or $S$), is it feasible to instantiate the unknown element such that query $Q$, when executed on the data $D$ satisfies the statistic $S$?

We claimed that multiple data management problems can be abstractly formulated as problems within this class. Chapters 3 to 7 present solutions to five such problems.

- **Part I: Missing Statistics**
  - In Chapter 3, the query $Q$ and data $D$ are provided. The statistic to obtain is the cardinality of various intermediate nodes in the query evaluation plan, *during query execution*.
  - In Chapter 4, the query $Q$ is a subexpression of the original query that forms a join pipeline. The data $D$ is the set of input relations to the pipeline. The statistic to obtain is the intermediate cardinality along all possible alternative orders of the pipeline that have the same outer relation.

- **Part II: Missing Queries**
In Chapter 5, the data $D$ is given, and the statistic $S$ is specified as the output cardinality of the query. The objective is to refine a query $Q'$ with user input to obtain a new query $Q$ that satisfies the output cardinality.

In Chapter 6, the data $D$ is provided, and the statistic $S$ is the cardinality of multiple intermediate subexpressions of a query template. The objective is to find an instantiation $Q$ of the query template that satisfies all the cardinality constraints defined by $S$ at the intermediate subexpressions when executed on $D$.

Part III: Missing Data

Finally, in Chapter 7, we are given multiple queries $Q_1, \ldots, Q_k$ which belong to the same query template. Each query is associated with cardinality and distributional constraints on intermediate subexpressions. The objective is to obtain a single test database $D$ such that each of the queries satisfies the associated statistical constraints on intermediate subexpressions when executed on $D$.

Each of these abstract problems is motivated by a real database management problem: monitoring (Chapter 3), adaptivity (Chapter 4), refinement (Chapter 5), and testing (Chapters 6 and 7).

We note that the definitions of “statistics” varies from problem to problem under consideration. This depends on the needs of each problem. For example, our work on query refinement (Chapter 5) focused on output cardinality of a single query, while our work on test database generation (Chapter 7) focused on the cardinality and data distributions at all intermediate nodes over multiple queries. Broadly speaking, our work in Part I focuses on estimating cardinality, but utilizes observed data distributions to obtain the cardinality at operators. Similarly, the techniques in Part II focus specifically on satisfying cardinality constraints, but also utilize samples to obtain data distributions for estimation. In Part III, the problem statement explicitly includes both cardinality and data distributions.

We next state the contributions of, and lessons learnt from the work presented in this thesis.

8.1 Contributions

Monitoring

Part I (Chapters 3 and 4) of this thesis demonstrates the feasibility and utility of overlaying statistics estimators on query evaluation plans. The key insight in our techniques is to exploit properties of various operators in a query evaluation plan to obtain the desired set of cardinality estimates. We demonstrate how to construct an appropriate set of histograms at blocking operators such as sorting and hashing. In addition, we present techniques to combine the histograms with random samples from specific inputs, to obtain the desired set of cardinality estimates.
8.1. Contributions

We demonstrate the utility of monitoring query execution through applications in progress estimation through the *ConEx* system in Chapter 3. Our system provides accurate cardinality estimates for various relational operators, with minimal overhead.

We also present applications of our monitoring framework to adaptive query processing in Chapter 4 as illustrated next.

Adaptivity

Chapter 4 of this thesis presents — to the best of our knowledge — the first adaptive join reordering technique (*XS*) that does not make cardinality estimation errors due to inter-table independence assumptions. As a consequence of this principled estimation method, *XS* can provide confidence guarantees on the quality of the reordered query plan.

The key technical insight in *XS* is that query evaluation plans of relational operators maintain too much state to allow continuous full plan adaptivity. Instead, *XS* exploits the structure of the query plan, identifying a moment of symmetry in the execution of a join pipeline to perform estimation. At this point, the inner inputs of the join pipeline have been processed, while the outer input remains untouched. Moreover, the pipeline has minimal internal state. Consequently, *XS* can reorder the inner inputs without affecting the correctness of the join pipeline.

A major enabler behind *XS* is the extension of *ConEx* to enable a lightweight estimation framework for all possible alternative orders of the join pipeline. We introduce the concept of simulation of execution of an alternative order of the pipeline. We also demonstrate how to simultaneously simulate the execution of multiple pipelines by maintaining dependency information in a join lattice.

Our results in Chapter 4 demonstrate that *XS* can reduce query execution times significantly, with minimal overheads.

Refinement

Chapter 5 of this thesis presents a novel take on the many/few answers problem. The key observation we make is that capturing user preferences by identifying a scoring function is a difficult task. Additionally, we note that scoring based approaches are not easy to adapt to the few answers problem, where the query is to be expanded.

Our contribution in this chapter is a new framework: *Stretch ‘n’ Shrink (SnS)*, that refines the queries with too many or too few answers by modifying the selection predicates. *SnS* explicitly incorporates user feedback in designing the new refined query through an interactive and iterative refinement procedure. We present refinement techniques for queries with predicates on categorical and numerical attributes, and demonstrate the utility and feasibility of our approach.
Testing

Finally, Chapters 6 and 7 provide new approaches towards testing the performance of database systems. The key observation in these chapters is that synthetic query and data generation techniques are operate independently of each other. Therefore, satisfying test cases that specify execution-time properties on intermediate data is not possible. For example, given a test query and a test database generated independently on the same schema, there is no guarantee that the query will not produce an empty result when executed on the data.

In Chapter 6, we present the \textit{TQGen} framework for targeted query generation. \textit{TQGen} assumes the data is given, and generates a test query that satisfies a given set of cardinality constraints in a best-effort fashion. We prove the hardness of the targeted query generation problem. However, our experimental evaluation suggests that the best-effort semantics of \textit{TQGen} work well in practice.

In Chapter 7, we invert the problem, and present the \textit{TDGen} framework for targeted database generation. Like \textit{TQGen}, \textit{TDGen} supports cardinality constraints on intermediate subexpressions. However, it also supports distributional constraints on relevant attributes as well. In addition, \textit{TDGen} supports generation of data satisfying multiple test queries simultaneously. \textit{TDGen} provides best effort semantics for distributional constraints, while satisfying cardinality constraints exactly.

Together, \textit{TQGen} and \textit{TDGen} provide tools for testing the performance of database systems in a targeted fashion.

8.2 Future Directions

Monitoring Parallel Query Plans

The statistics estimation framework presented in chapters 3 and 4 assumes the query evaluation plan is executing as a single process. In practice, data management systems often execute queries in parallel across multiple machines [41]. One of the major differences that arise due to this is that relations may no longer be processed in order. For example, a sort-merge join on a uniprocessor sorts one relation, and then the next relation. As a consequence, \textit{ConEx} could build a histogram on the first relation being sorted, and probe it using the second relation. In contrast, in a parallel query evaluation plan, the sorting phases for both relations may proceed in parallel. In addition, with the data spread across multiple machines, it is no longer feasible to generate histograms for the entire table during query execution time. Instead, statistics need to be obtained at the level of each chunk of the data, and aggregated appropriately. We have recently initiated preliminary work that explores the problem of monitoring parallel query plans in the context of map-reduce environments [109].
Applications of Query Monitoring

Chapter 3 of this thesis proposed a framework for monitoring the progress of query execution. Chapter 4 presented an extension and an application of the framework to the problem of adaptively reordering join pipelines. A natural question that arises is whether monitoring can be applied to other data management problems.

As an example, there has been increasing interest [24, 30] in the problem of suspending and resuming the execution of database queries. Implementing any pre-emptive scheduling technique inside a database system would require a feature for suspending and resuming queries. However, database queries typically have a large state and therefore suspending them to disk is an expensive operation. A query monitoring framework can provide online information as to the expected behaviour of the query, and enable a query suspend/resume technique to make informed decisions about whether it should (a) Roll the query back to the previous low memory state point in the query (b) Suspend the query, writing the state out to disk or (c) Allow the query to execute further until the next point where its memory state is minimal.

More generally, resource allocation and system tuning problems could potentially benefit from monitoring the current state, and predicting the future behaviour of the query. In such problems, the objective is to adaptively allocate resources (e.g. memory, processors, buffer and temp space, and even indices and statistics) based on the current (and future) properties of the currently executing workload. Applications of the ConEx query monitoring framework to such problems remains a future direction worth exploring.

Unifying Query Reoptimization

Query reoptimization techniques for relation databases come in many forms; From local adaptivity as encapsulated in the systems of Li et al. [87], Rio [11], and XS (Chapter 4), to global reoptimization such as the Re-Opt [82] and POP [98] frameworks. Every query re-optimization technique, performs a risk opportunity tradeoff. XS for instance, takes a low-risk approach towards re-optimization, and consequently considers only a subset of possible alternative query execution plans. Techniques like POP [98] have the scope of performing bigger reoptimizations to the query evaluation plan; however, they are also more likely to hurt query performance as well. We envisage techniques like XS working in tandem with global reoptimization frameworks (such as POP), with a decision making component which defines the risk-opportunity tradeoffs associated with each framework. Devising such a framework is a challenge that remains open.

Test Query and Data Generation

TQGen (Chapter 6) and TDGen (Chapter 7) are query and data generation frameworks that attempt to satisfy test cases that specify statistical properties at execution time. This is a relatively
new area of research, and consequently there are several future threads to explore. Examples include:

- Extending the class of operators covered by \textit{TQGen} and \textit{TDGen}. Both \textit{TQGen} and \textit{TDGen} support queries with conjunctive selections, projections, equijoins, and a single aggregation operator. Database systems support many other operators, such as non-equijoins, outerjoins, distinct, union, intersection and set difference operators. Moreover, database systems also support sub-queries, which cannot be handled by \textit{TQGen} and \textit{TDGen}.

- Extending \textit{TQGen} to simultaneously handle multiple test cases. Unlike \textit{TDGen}, \textit{TQGen} operates on a per-test case basis. However, testing plans typically consist of multiple test queries. Batching the processing of \textit{TQGen} could potentially reduce query generation times significantly.

- Extending \textit{TDGen} to support a larger class of queries. Potential extensions include joins with neither input being a key, other relational operators, and removing the constraint on an attribute being relevant to at most one operator.

- Extending \textit{TQGen} and \textit{TDGen} to support cardinality constraints specified as ranges i.e cardinality between 100 and 200 million tuples, or cardinality at least 1 million tuples. Formally analyzing the hardness of these variants of the problem remains open.

- Data Transformations. \textit{TDGen} currently generates a test database from scratch, which may be an expensive operation for large data. Instead, it is worth exploring how to modify an existing test database (through insertions and deletions) in a minimal fashion, so as to satisfy the constraints imposed by the test case.

- Dataflow Programs. Recent work [110] on generating example data for dataflow programs (such as Map-Reduce) highlights the difficulty that programmers have in writing correct programs. A similar concern is the efficiency of the program; dataflow programs essentially require the programmer to write a query evaluation plan by themselves. Extending \textit{TQGen} and \textit{TDGen} to this space would enable programmers to test the performance of their program in a targeted fashion.


