ONLINE PHYSICAL DESIGN AND MATERIALIZATION IN SCALABLE DATA MANAGEMENT

by

Jiang Du

Graduate Department of Computer Science
University of Toronto

2019
The technique of materializing intermediate results and reusing these results to answer future queries is applied widely in data management because it can greatly improve the performance of real-life workloads. Although it has proven to be a crucial performance factor, the physical design of data is usually considered independent of materialization. One problem of tackling physical design and materialization independently is that it neglects the correlation between these two techniques and may produce suboptimal solutions. The access patterns of real-life workloads typically evolve frequently, and the evolving access patterns bring the challenge of automatically adapting the physical design to the workload. An effective approach to solve this problem is an online physical design, which does not assume prior knowledge of the workload but adapts the design based on the history of the workload. In this thesis, we discuss a general problem of combining physical design and materialization to improve workload performance. We prove our solution to the general problem is broadly applicable to instances of this problem by systematically investigating: (1) partition and re-partition materialized views to honor the access patterns of non-update workloads; and (2) allocate data dynamically between streaming engines and analytical engines in a federated system for hybrid workloads consisting of data ingestion, transactions, and analysis. We then extend our research by showing a preliminary study on how to apply our solution to a third instance of this problem: tune data co-location and materialization for non-update workloads. Our techniques work in an online fashion and thus are capable of adapting the design automatically to evolving workloads. We demonstrate that our techniques of integrating physical design and materialization significantly outperform approaches that solve them independently and thus greatly improve the workload performance.
Acknowledgements

Firstly I would like to thank my advisor, Renée J. Miller, who has been supportive during my PhD program. I have benefited enormously from her remarkable vision, experience and knowledge. She has been a great mentor for guidance. I am grateful for the long hours that she devoted to reading and correcting my drafts. She helped me to build connections with the collaborators outside the University of Toronto. It is crucial to the success of my research.

I would also like to thank the members of my PhD committee (Boris Glavic and Ken Pu) for reading my thesis and providing feedback. I especially want to thank Boris Glavic, who has been involved in all of my research projects. It was Boris who helped me to establish the foundation of my research. From him I have learned not only how to conduct research, but also how to think and communicate through writing. This work would not have been possible without his knowledge and assistance. I also thank Ashraf AboulNaga for serving as the external examiner of my thesis, and for coming to Canada from Qatar to attend my thesis defense in person. Ashraf’s insightful feedback helps me to reach a higher level of understanding of the research topics.

From 2014, I was fortunate to start working with Stan Zdonik and Nesime Tatbul. They are not only outstanding researchers, but also incredibly warm and generous people. They have always been supportive of my research. It was their encouragement that helped me to get through the most difficult time in my PhD studies. In 2013 and 2014, I had the opportunity to work with Wei Tan and Liana Fong in IBM Watson. They are great mentors. DeepSea was published under their guidance. I also thank Hao Tan for helping me to build and evaluate the framework in Chapter 5.

Last but absolutely not the least, I want to thank my parents, Yongshen Du and Jingze Gao, for being with me despite the distance. My wife, Jing Wu, it would have been impossible for me to accomplish this without her love and support.
# Contents

## 1 Introduction

1.1 The Problem of Combining Physical Design and Materialization ........................................ 3  
1.2 Progressive Partitioning of Materialized Views ................................................................. 4  
1.3 Online Data Placement for Polystore Ingestion ................................................................. 5  
1.4 Online Data Co-Location and Materialization ................................................................. 6  
1.5 Contributions ...................................................................................................................... 7  
1.6 Outline ............................................................................................................................ 9  

## 2 The Problem of Combining Physical Design and Materialization ..................................... 10

2.1 Introduction ...................................................................................................................... 10  
2.2 Properties of real-life workloads ...................................................................................... 12  
2.3 Related Work ................................................................................................................. 14  
2.4 Preliminaries .................................................................................................................. 16  
2.5 Problem Statement ........................................................................................................ 18  
2.6 Solution Overview ......................................................................................................... 18  
2.6.1 Match Design .............................................................................................................. 20  
2.6.2 Select Rewriting ......................................................................................................... 21  
2.6.3 Determine Candidates ................................................................................................. 21  
2.6.4 Select Design ............................................................................................................. 22  
2.6.5 Update Statistics ......................................................................................................... 22  
2.7 Applicability .................................................................................................................. 23  
2.7.1 Progressive Partitioning of Materialized Views ......................................................... 24  
2.7.2 Online Data Placement Design for Polystore Systems ............................................. 24  
2.7.3 Online Data Co-Location and Materialization ......................................................... 24  

## 3 Progressive Partitioning of Materialized Views ............................................................... 26

3.1 Introduction ...................................................................................................................... 26  
3.2 Related Work .................................................................................................................. 30  
3.3 Preliminaries .................................................................................................................. 31  
3.4 Problem Statement ....................................................................................................... 33  
3.5 Solution Overview ........................................................................................................ 35  
3.6 View and Partition Candidates ...................................................................................... 36  
3.6.1 View Candidates ......................................................................................................... 37  
3.6.2 Partition Candidates ................................................................................................. 38
3.7 Views and Partition Selection ................................................................. 39
  3.7.1 Cost and Benefit Model ................................................................. 40
  3.7.2 Filtering View and Partition Candidates ........................................... 44
  3.7.3 View and Fragment Selection .......................................................... 45
3.8 View and Partition Matching .............................................................. 46
  3.8.1 A Sufficient Condition for Matching ................................................ 47
  3.8.2 Partition Matching ...................................................................... 47
  3.8.3 Indexing Partitioned Views .............................................................. 48
  3.8.4 Updating View and Partition Statistics .............................................. 49
3.9 Implementation .................................................................................... 49
3.10 Evaluation ........................................................................................... 51
  3.10.1 Workload for a Real-Life Application .............................................. 52
  3.10.2 Adaptive and Progressive Partitioning .............................................. 54
  3.10.3 Exploitation of Fragment Correlations ............................................. 58
  3.10.4 Overlapping Partitioning ............................................................... 58
3.11 Conclusions ......................................................................................... 62

4 An Online Data Placement Design for Polystore Systems ............................ 64
  4.1 Introduction ....................................................................................... 64
  4.2 Related Work .................................................................................... 68
  4.3 Background ....................................................................................... 70
    4.3.1 BigDAWG Polystore .................................................................... 70
    4.3.2 S-Store ..................................................................................... 71
  4.4 Motivation ......................................................................................... 72
  4.5 The Data Placement Problem ............................................................. 76
    4.5.1 Preliminaries ............................................................................. 76
    4.5.2 Problem Statement ................................................................... 81
  4.6 Solutions ........................................................................................... 82
    4.6.1 Solution overview ..................................................................... 82
    4.6.2 Cost model ............................................................................... 85
    4.6.3 Data placement procedures ....................................................... 93
  4.7 System Prototype and Experimental Evaluation ................................... 94
    4.7.1 Evaluation Setup ....................................................................... 95
    4.7.2 Results ..................................................................................... 98
  4.8 Conclusion ......................................................................................... 103

5 Online Data Co-Location and Materialization ............................................. 105
  5.1 Introduction ....................................................................................... 105
  5.2 Related Work .................................................................................... 110
  5.3 Preliminaries ..................................................................................... 112
  5.4 Problem Statement .......................................................................... 115
  5.5 Solution Overview ............................................................................ 115
    5.5.1 Co-location .............................................................................. 117
    5.5.2 Assumptions ............................................................................. 118
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6 Example Workload</td>
<td>118</td>
</tr>
<tr>
<td>5.7 View and Co-Location Candidates</td>
<td>124</td>
</tr>
<tr>
<td>5.7.1 View and Co-Location Statistics</td>
<td>125</td>
</tr>
<tr>
<td>5.7.2 View Candidates</td>
<td>125</td>
</tr>
<tr>
<td>5.7.3 Co-Location Candidates</td>
<td>126</td>
</tr>
<tr>
<td>5.8 View and Co-location Matching</td>
<td>127</td>
</tr>
<tr>
<td>5.8.1 View matching</td>
<td>128</td>
</tr>
<tr>
<td>5.8.2 Co-location matching</td>
<td>129</td>
</tr>
<tr>
<td>5.9 View and Co-Location Selection</td>
<td>130</td>
</tr>
<tr>
<td>5.9.1 Cost Model</td>
<td>130</td>
</tr>
<tr>
<td>5.9.2 Calculating the cost of views and co-located relations</td>
<td>131</td>
</tr>
<tr>
<td>5.9.3 Calculating the (potential) benefit of views and co-located relations</td>
<td>131</td>
</tr>
<tr>
<td>5.9.4 View selection and plan selection</td>
<td>133</td>
</tr>
<tr>
<td>5.9.5 Updating cost and benefit</td>
<td>137</td>
</tr>
<tr>
<td>5.10 Preliminary Experimentation</td>
<td>138</td>
</tr>
<tr>
<td>5.11 Conclusions</td>
<td>139</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>140</td>
</tr>
<tr>
<td>6.1 The Problem of Combining Physical Design and Materialization</td>
<td>140</td>
</tr>
<tr>
<td>6.2 Progressive Partitioning of Materialized Views</td>
<td>141</td>
</tr>
<tr>
<td>6.3 Online Data Placement for Polystore Systems</td>
<td>141</td>
</tr>
<tr>
<td>6.4 Online Data Co-Location and Materialization</td>
<td>142</td>
</tr>
</tbody>
</table>

Bibliography 144
Chapter 1

Introduction

In the past decade, we have witnessed rapidly progressing data management techniques developed for exploding data volumes. Computation frameworks such as MapReduce [32], Dryad [53], and Spark [86] have been proposed to address challenges such as the fault tolerance and the scalability for shared-nothing distributed systems that are built on top of thousands of commodity computers.

Materialization is an effective technique to improve the performance of real-life workloads. It materializes intermediate query results and reuses these results (known as materialized views) to answer future queries that share these intermediate results. Materialization has been studied extensively in the literature [27, 49]. In recent years we have observed the efforts to extend materialization from standalone systems to shared-nothing distributed systems [38, 73]. However, these proposed systems neglect some important properties of real-life workloads (e.g., frequently evolving access patterns, and the correlation between physical design and materialization) executed in a distributed system. Considering these properties in the design phase has raised new challenges accordingly.

The first challenge is how to combine physical design with materialization to improve the performance of the execution of the workloads. Physical design has been proven to be an important performance factor because of the massive data volume of today’s data management. Materialization is a technique that can also greatly improve the performance of the workloads.
Combining proper physical design and materialization becomes natural for even better performance. Although both techniques have been studied separately for a long time [61, 71], combining them is not a trivial task, as we will see in the following chapters.

The second challenge is how to make our design techniques adapt to the access patterns of the workloads in an online fashion. Conventional data analysis is relatively static, but today’s workloads change much more frequently [63]. An offline approach, i.e., optimizing the design for a representative workload, assuming the access patterns of the workload do not change much over time, does not work in this setting. An online approach that does not assume a prior knowledge of the workload, but adjusts the design based on the history of the workload, is more preferable.

The third challenge is how to effectively apply our design techniques to both non-update workloads and hybrid workloads (containing data ingestion, transactions, and analysis queries). We have observed the appearance of Polystore systems (e.g., [39]), where a number of database engines designed for different data models are connected together in a federated fashion. Such a federated system is capable of running different types of computations such as streaming, online transaction processing (OLTP) and online analytical processing (OLAP). Data is automatically migrated between the engines in the system. Applying our design techniques to a Polystore system allows efficient data migration to improve the overall performance of a hybrid workload. All these challenges suggest that a new framework for physical design and materialization is required to consider the massive data volume, the ad hoc queries (meaning specialized or non-general purpose) in real-life workloads, the evolving nature of workloads, and the complexity of heterogeneous database engines.

This thesis argues that a framework that deeply couples physical design and materialization in one unified optimization process can solve the above challenges effectively. The proposed framework has a number of advantages over the current systems.

1. Considering physical design and materialization together has a better chance of finding optimal (or near-optimal) solution(s). When they are considered separately as independent processes, the correlation between them is neglected. As we will see in the following
chapters, neglecting the correlation often leads to the select of suboptimal designs.

2. Tuning physical design and materialization in an online fashion is important because the access patterns of the workloads change frequently. A recent work shows that in a real-life workload, after seeing 80% of the queries in the workload, the remaining 20% contains 57% new queries [63].

3. Combining physical design and materialization in an online fashion improves the overall performance of hybrid workloads including both read and write operations.

We implemented our techniques in a stack of open source systems.


2. Implemented in the optimizer of BigDAWG [39] and S-Store [66] for a Polystore system.


**Thesis Statement:** A framework integrating physical design and materialization in an online fashion can effectively improve the performance of real-life workloads.

### 1.1 The Problem of Combining Physical Design and Materialization

Physical design is a set of techniques such as indexing and data partitioning that study how data is stored and organized physically in a data management system. Physical design is a major factor for the performance of database operations. Tuning physical design automatically has been studied for a long time [11]. Materialization, another important factor for the database performance, is also a common technique. Materialization may be done at design-time or at runtime, e.g., a query optimizer may select which query results to materialize based on the knowledge of the history of a workload. The questions of what to materialize, when to materialize, and when to use a view to answer a query have been studied extensively [8].
Physical design and materialization are often considered as independent techniques. We argue that they are correlated, and tuning them separately often yield suboptimal solutions. To combine physical design and materialization, it is important to understand the requirements of today’s real-life workloads. Recent research [15, 28, 37, 63] and our study of the Sloan Digital Sky Survey (SDSS) [5] (a real-life analytical workload) show several properties of today’s real-life workloads:

1. Workloads usually access data in a non-uniform fashion.
2. The access patterns change over time.

The above observations lead to the following requirements for today’s real-life workloads:

1. The design process should integrate physical design and materialization.
2. The design must consider the non-uniform access pattern of the workload.
3. The design must be self-adjusted as the workload evolves.

Based on the above requirements, we formalized the general problem of combining physical design and materialization to improve workload performance and developed a broadly applicable solution to the problem. Our solution integrates physical design and materialization in an online fashion, and adjusts the design automatically according to the recent operations in a workload, assuming that these operations are indicative of the future operations in the workload.

1.2 Progressive Partitioning of Materialized Views

We first study an instance of the general problem in which the workload is non-update (also known as analytical workloads). As we have introduced in the above section, materialization has been studied for a long time. In contrast, the physical design of views and how to adjust this design to the current workload has received less attention. In modern SQL systems built on-top of distributed execution engines (e.g., Hadoop [1]), because of the massive data
volume, issues of physical design (including partitioning of large files) are paramount to the performance of the system. We will show that partitioning a materialized view into a set of fragments and using only necessary fragments to answer new queries can greatly improve performance. An important difference of partitioning materialized views and partitioning base tables is that materialized views are augmented data structures, and we can always recompute them from the base tables. In contrast to partitioning base tables for which we always have to keep all fragments to not lose data, we can selectively decide which fragments of the partitioned materialized views to keep. This opens up new optimization opportunities, e.g., materializing only the fragments that are predicted to bring benefits for future queries.

In addition to the common properties that we find for real-life workloads that we introduced in the previous section, our study of SDSS also shows that the ranges of domain values that are accessed frequently usually have neighbors that are also accessed frequently. All these properties suggest that: (1) the partitioning of materialized views should not be static (e.g., equi-depth or equi-width), but rather should be adapted to the workload; (2) the partitioning must be progressive (partition and re-partition the materialized views) to account for the changes of the access patterns of the workload; and (3) when we conduct fragment selection (deciding what fragments should be materialized and what should not), we must consider the correlation among the fragments.

We adapted our general solution to this problem, and proposed a partitioning technique for materialization suited for the non-update workloads with these properties. The technique is embedded in the query optimizer, progressively adapting the design (partitioning and re-partitioning the materialized views) to the access patterns of a non-update workload.

### 1.3 Online Data Placement for Polystore Ingestion

Secondly, we study another instance of the general problem in which the workload is hybrid and contains both read and write operations. In recent years, a number of new database markets such as data warehousing, stream processing, and non-structure data processing have greatly
diversified the requirements for database systems. As a result of this change, Stonebraker et al. [80] claim that there is no one-size-fits-all database system that can meet the challenges for today’s data processing anymore, and call for a complete rewrite to systems customized for different requirements. Database engines such as H-Store (transaction processing) [57], S-Store (stream processing) [24], and TileDB (array database) [6] have appeared, and a federated architecture known as Polystore has been proposed to enable query processing over these databases (e.g., BigDAWG [39]).

In a Polystore system such as BigDAWG, the streaming engine is the entry point for data ingestion [65]. A modern streaming engine such as S-Store [24, 66] guarantees transactional safety of data ingestion. S-Store contains a limited storage space in memory, which can be used to temporarily stage ingested data. In order to optimize the performance of a hybrid workload consisting of data ingestion, transaction, and analysis, this storage must be used efficiently. An important question therefore is to design the placement of ingested data, and to migrate data effectively between S-Store and the analytical engine.

We modeled the general problem of combining physical design and materialization (Section 1.1) into the problem of the data placement design in Polystore, and developed a cost model and an online approximated algorithm based on our solution to the general problem to improve the performance of a hybrid workload.

1.4 Online Data Co-Location and Materialization

Thirdly, we show preliminary research on a third instance of the general problem. Data placement has been proven to be a major factor for the performance of the workloads executed in shared-nothing distributed systems [9, 72]. In a shared-nothing distributed system, tables are usually partitioned across nodes. When two tables are joined by a query, if the data is partitioned such that any two tuples having the same join attribute values are placed on the same node (co-located), the join can be executed locally. If this is not the case, we will have to repartition the data to ensure co-location. This process is often called data shuffling. A join
that requires data shuffling is called shuffle join [9]. Shuffle joins often incur heavy remote
communication (data movement) among a large number of nodes in the cluster. As a result of
the remote data movement, shuffle join is usually an expensive operation. On the other hand, if
the tables in a join operation are co-located, then the overhead of data shuffling can be avoided.

A large amount of research has studied how to co-locate data [35, 40] and how to decide
what data should be co-located [30, 44, 59, 63]. Data co-location design is often modeled
as a graph partitioning problem, where the tuples are the nodes in the graph, and join paths
are the edges. The design can then be modeled to computing graph partitioning. As a result,
it becomes an expensive process that is non-scalable in an online fashion. Meanwhile, data
colocation has a tremendous impact on how we conduct materialization (when to materialize,
what to materialize, and how to reuse a view to answer a query). Consider the case where two
tables are co-located. This data layout can affect our decision of materializing the join result
of the two tables. If materializing the join result is expensive, we may choose not to do so,
but rather to compute the join result from the co-located base tables because the join can be
executed locally (thus be an inexpensive operation). As we see from this example, separating
physical design and materialization into two independent processes can yield suboptimal query
plans.

Again, we modeled the general problem of combining physical design and materialization
to the problem of integrating data co-location and materialization. Our cost model and the ap-
proximated, scalable, and online algorithm is adapted from the solution to the general problem.

1.5 Contributions

In this thesis, we studied the problem of online physical design and materialization for scalable
data management. We systematically investigated two instances pertaining to this problem,
namely: (1) progressive partitioning of materialized views, and (2) online data placement for
Polystore systems. We then extend our study by preliminary research on a third instance of the
general problem: integration of data co-location and materialization. Below we summarize our
main contributions.

The Problem of Combining Physical Design and Materialization

- We formally defined the general problem of combining physical design and materialization to improve real-life workload performance.
- We developed a solution that is broadly applicable to such a problem.

Progressive partitioning of materialized views

- We proposed the first algorithm to progressively partition materialized views. The partitioning adapts online to the changes in a workload.
- Based on our study of real-life workloads, we presented a novel cost-benefit model for view fragments that takes the correlation among fragments of a view into account.
- We presented DeepSea, an implementation of our techniques in Hive [82].
- We demonstrated DeepSea’s effectiveness, using a query workload consisting of the query templates from BigBench [42] and modeled on a real-life workload from SDSS [5].

Online data placement for Polystore systems

- We proposed the first online data placement design for a Polystore system that contains a streaming engine and an analytical engine.
- We developed a cost model that considers a number of key factors for the performance of a hybrid workload.
- We implemented our cost model and a scalable online algorithm in the optimizer of an open source Polystore system (BigDAWG) and a streaming engine (S-Store).
- We evaluated our techniques on synthetic workloads that are modeled on real-life workloads.

Data co-location and materialization

- We proposed to co-locate data as a byproduct of query processing.
- We integrated data co-location and materialization in one unified framework to improve the workload performance.
- We designed a cost model for the unified framework, and developed an online scalable algorithm based on this cost model to adapt the design to the changes of the workload.
CHAPTER 1. INTRODUCTION

1.6 Outline

This document is organized as following. In Chapter 2, we discuss the general problem of combining physical design and materialization to improve workload performance. In Chapter 3, we study how answering queries by reusing materialized views can benefit from properly partitioning and re-partitioning the views. In Chapter 4, we present an online data placement design for the streaming engine and the analytical engine in a Polystore system. In Chapter 5, we propose a preliminary online integrated design of data co-location and materialization for shared-nothing distributed systems. In Chapter 6, we conclude the work in this thesis and propose new avenues for future research.
Chapter 2

The Problem of Combining Physical Design and Materialization

2.1 Introduction

In this chapter, we study the problem of combining physical design and materialization to improve the performance for real-life workloads.

In this thesis, physical design refers to how data is organized and stored physically in a database system. Physical design is one of the most important factors for database performance. For instance, an index structure is a physical design that may speed up record lookup, meanwhile it requires extra cost for record update; whether an index structure improves the performance depends on the schema and the workload. Physical design can be performed by a human, e.g., a database administrator (DBA), when the schema of the database and the workload is not complex. However, when the complexity of the schema and the workload increases, human design often becomes impractical. Automatic physical design usually considers multiple factors automatically and thus is capable of conducting designs for complex schemas and workloads. Research on automatic tuning of physical design has a long history [10, 11, 12, 13, 20, 71, 75]. In this thesis, we study how to integrate physical design with materialization automatically to improve workload performance.
Materializing intermediate results and reusing these results to answer future queries is called materialization. Materialization is a common technique to improve the performance of workloads [61]. The questions of what to materialize, when to materialize, and when to use a view have been well studied. In modern SQL systems built on-top of distributed dataflow engines (e.g., Hadoop [1]), intermediate results are often materialized for fault tolerance purposes and these results can be utilized as materialized views to answer future queries [38]. While each of these techniques has been studied intensively, we are the first to study the combination of physical design and materialization.

Physical design and materialization are often correlated. As we will show in the following chapters, a design process that consider these two techniques independently usually provides suboptimal solutions. For instance, creating materialized views without considering the data access patterns can lead to inefficient use of the view pool, because we would not be able to store only the most beneficial partitions (compared to storing a whole view) in the pool (Chapter 3); designing data placement alone without considering the differences between caching (keeping multiple copies of the same data as a materialization technique in a federated system) and anti-caching (keeping a single copy of the data in the system) can lead to a suboptimal solution for a hybrid workload containing both streaming operations and OLAP queries, when data migrations between database engines are required (Chapter 4).

**Requirement 1** The design process should integrate physical design and materialization.

Next we investigate several common properties of today’s real-life workloads so that we can address the requirements of these properties, and then we will discuss our strategy of combining physical design and materialization automatically in order to meet these requirements.
2.2 Properties of real-life workloads

Research has shown several common properties of today’s workloads. We investigated a real-life workload and obtained similar conclusions. In this section we discuss these properties and their implications to the requirements, and briefly describe our solution to these requirements.

Non-Uniform Distribution of Access. Workload access is usually not uniform. In recent benchmarking research [28], the authors studied real-life OLTP workloads and found that it is common that the data access follows Zipfian distributions.

We also studied a real-life non-update OLAP workload. Figure 2.1 shows the access distribution for a real analytic workload over the Sloan Digital Sky Survey dataset (SDSS) [5]. The figure shows the selection ranges on attribute \( ra \) of table \( PhotoPrimary \) for queries submitted to SDSS between March 8, 2010 and March 8, 2011. Note that there are ranges that are rarely queried and others that are very frequently queried.

When we consider combining materialization and data partitioning (a physical design technique) for such a workload, a static design solution such as equi-width partitioning will not work well for the workload; instead the solution must be adaptive to the workload, i.e., when using data partitioning for the design, we must consider what fragments (partitions) are ac-
cessed more often than others, and optimize the partitioning for these fragments.

Requirement 2 The design must adapt to the characteristics of the workload.

Evolving Access Patterns. Workload access pattern usually evolves. In addition to being non-uniform, real-life workloads are not static, and access patterns may evolve over time. It has been shown that today’s real-life workloads evolve frequently [15, 37, 63].

Continuing with the analysis of the workload described in Figure 2.1, Figure 2.2 shows how the selection ranges of SDSS queries over attribute $ra$ of table PhotoPrimary evolve over the sequence of the first 10,000 queries containing such a selection, starting from March 8, 2010.\(^1\) The figure shows that the first 3,000 queries focus mainly on the range between 200 and 300 degrees. Later in the workload, a large number of queries focus on values around 100 degrees. This implies that the combination of physical design and materialization must consider the evolving access patterns, and automatically adjust itself as the workload evolves.

\(^1\)The vertical line near query 1,000 means that one or more queries have selected the whole domain of attribute $ra$. 

Figure 2.2: Evolution of selection ranges on SDSS
Now we discuss our strategy of automatically combining physical design and materialization to improve the workload performance, motivated by the above two properties of real-life workloads.

**Online Design Selection.** A design selection algorithm that is based on a query workload is called *adaptive*. Adaptive (or workload-aware) design for physical design and materialization may be done at design-time or at runtime. That is, either a complete workload is given and the design selection algorithm determines which views to materialize and how to perform physical design offline, or the algorithm works in an *online* fashion, making decisions based on the history of operations that have been processed so far. While online materialized view selection has been studied [73], we are the first to consider the online adaptation of design choices for combining physical design and materialization. In the next section we will investigate related work and show why the current research fails to fulfill the requirements for today’s real-life workloads.

### 2.3 Related Work

There are several lines of work related to our approach: answering queries using views; reusing intermediate query results; and (online) self-tuning techniques for physical database design.

**Answering Queries Using Views.** Answering queries using materialized views has been studied intensively [8, 61]. Given a set of views and a query, computing the least expensive plan for the query using the views is computationally hard, because query containment checks are required to determine whether a query can be computed from a view. Query containment for bag semantics (SQL) is undecidable, even for restricted query classes (union of conjunctive queries). As a consequence, practical approaches for *logical matching* (i.e., determining whether a view can be used to answer a query independent of the query syntax) usually apply...
sufficient conditions for matching that are decidable or even in PTIME [46, 87]. Goldstein and Larson [46] present a lightweight algorithm that is integrated with a transformation-based optimizer and uses a cost model to determine the best rewriting. The research only considers how to answer queries by using views. The physical design of the data and the correlation between physical design and materialization is not considered (Requirement 1 in Section 2.1).

**Reusing Intermediate Results.** Although materialization has been studied extensively for relational databases [8, 49, 61], distributed systems such as Hadoop have different characteristics that need to be explored and exploited. ReStore [38] materializes intermediate results of MapReduce jobs for reuse in future queries. Perez and Jermaine [73] exploit salient features of SQL-on-Hadoop systems including immutable data, abundant storage to accommodate materialized views, and excessive materialization of intermediate results that enables generating materialized views as a by-product of answering queries. The approach optimizes queries to produce intermediate results that, if materialized as views, would improve performance for past queries. If past queries are indicative of future queries then this would result in a speed up for future queries. The Nectar system [47] caches and reuses results of DryadLINQ/Dryad computations. ReStore and Nectar only perform physical matching, i.e., a view matches a sub-query if they are computed using the same expression. Reuse of intermediate query results has also been studied for main memory DBMS such as MonetDB [54, 69]. This approach uses physical matching of operators except for selections where subsumption of range restrictions is considered, e.g., the result of a selection on \( A < 5 \) is a superset of the result of a selection on \( A < 3 \), and thus a query with selection \( A < 3 \) can be rewritten by using a materialized view whose selection is \( A < 5 \). Although there is work [73] that addresses challenges brought by evolving workloads by performing design in an online fashion, similar to the above line of work for answering queries using views, the current research in this line of work often lacks the consideration of physical design and the correlation between physical design and materialization (Requirement 1 in Section 2.1).

**Automated Physical Design.** Automated tuning [26] is a rich field including: partitioning [71, 75], index selection [13, 76], and materialized view selection [10, 11, 17]. Adaptive index
selection creates and drops indexes on-the-fly [13, 76]. Given a constraint on storage space, the idea is to monitor incoming queries and profile the performance gain for each index and then create the most promising ones. Adaptive materialized view selection [10, 11, 17] shares the same philosophy. Both index and materialized view selection use the DBMS optimizer’s cost model to evaluate the benefits of an index or view without actually creating it. Bruno and Chaudhuri [21] have explored online index selection that is 3-competitive. However, this bound only holds for single index candidates. The H2O system [14] supports multiple storage layouts, i.e., columnar, row and group of columns. At run-time, the system decides which layout to use for which part of the data, and continuously evolves the storage layout and data access strategy. Addressing the challenges in an online fashion is common in this line of work. However, materialization as an important design technique is often neglected (Requirement 1 in Section 2.1).

### 2.4 Preliminaries

We now introduce necessary concepts before we formerly define the problem studied in this thesis.

For a data management system, our goal is to adjust the design (a combination of physical design and materialization) to optimize (or near-optimize) the performance of a workload.

**Definition 2.1 (Workload).** A workload is a sequence of \( n \) operations \( Q_1, \ldots, Q_n \). Each operation \( Q_i \) can read or write a set of tuples. We use \( W \) to denote this workload.

In order to optimize the design for a workload, we take the workload history as an indicator of the characteristics of the workload. We tune our design according to the workload history, and assume the unseen operations in the workload follow the same access patterns in the workload history.

**Definition 2.2 (Workload History).** Given a workload \( W \), we use \( \mathcal{H}(i) \) to denote the workload history, which is the prefix of \( W \) up to the current query \( (Q_i) \).
Having introduced the workload and workload history, we are ready to introduce the design that we optimize for a workload.

**Definition 2.3** (Configuration). A configuration is materialized views and data structures of other physical design that we maintain for the system.

A configuration contains the data structures of a design before the execution of an operation. For instance, when we consider partitioning as a physical design when combining with materialization, the configuration consists of the fragments of materialized views that we maintain. We use $C_i$ to denote the configuration before the execution of $Q_i$. There is an upperbound for the size of a configuration $S_{max}$. The total size of the data structures of a configuration cannot exceed $S_{max}$. We assume there is no design at the beginning of $W$, i.e., $C_1 = \emptyset$. We compute a sequence of configurations $C_2, \ldots, C_n$ such that the workload performance is optimized subject to the size of each configuration does not exceed $S_{max}$.

In order to optimize the workload performance, we must define the cost of an operation in workload $W$. This cost is based on the current configuration.

**Definition 2.4** (Execution Cost). We use $\text{COST}_{EXE}(Q, C)$ to denote the cost of executing operation $Q$ in configuration $C$.

Besides the execution cost of an operation, the workload performance also depends on the cost of generating new physical designs and materialized views.

**Definition 2.5** (Configuration Cost). We use $\text{COST}(C_i, C_{i+1})$ to denote the configuration cost for $C_{i+1}$, i.e., the cost of creating new physical designs and materialized views that are not in $C_i$.

Comparing to the creating cost of a new data structure, we assume the removing cost is negligible, and thus when we compute the cost of a workload, we neglect the cost of removing a data structure from a configuration.
2.5 Problem Statement

We now define the problem addressed in this thesis: how to select a set of augmented data structures in the configuration (including physical designs and materialized views) in order to optimize the workload performance.

For an input of workload $W$ and $C_1 = \emptyset$, find a sequence of configurations $C = C_2, ..., C_n$ to minimize the total execution cost of the operations in $W$

$$\text{Cost}(W, C) = \sum_{i=1}^{n} \text{Cost}_{\text{EXE}}(Q_i, C_i) + \sum_{i=1}^{n-1} \text{Cost}(C_i, C_{i+1}) \quad (2.1)$$

We would like to point out that it is generally intractable to find an exact solution to this problem, because even a simplified version of this problem that considers only materialized view selection is computationally hard as we have introduced in the related work. Thus in this thesis we strive to find a scalable solution that provides approximated answers such that the design is optimized or near-optimized.

2.6 Solution Overview

As we have introduced in Section 2.1, we choose to solve the problem defined in Section 2.5 in an online fashion, i.e., instead of tuning the design for a representative workload assuming the characteristics of the workload do not change over time (also called the offline fashion), we tune the design based on the workload history, making the assumption that the workload history indicates future operations in the workload. Recall Requirement 3 in Section 2.1 (the design must evolve when the workload evolves), we choose the online approach because research has shown that the access patterns of today’s real-life workloads are often not static.

The online approach is difficult for several reasons. For each incoming operation $Q$, we must decide how to adjust the configuration without knowing the remaining sequence of queries from the workload. There is abundant literature for online algorithms that provide worst-case guarantees. An online algorithm is said to be $k$-competitive if its result is at most of a factor $k$ worse than the solution computed by an optimal offline algorithm (an algorithm which has
access to the whole input). However, the competitiveness factor of such algorithms for search space sizes encountered in our problem are too high to be of any practical relevance. Even if we were to consider the offline version of the problem, we cannot hope for an optimal solution because of the undecidability of query answering with views (it is not always possible to know what views could be used to answer a query and thus we do not know which views provide the best performance).

Given these constraints we strive for a principled yet scalable solution that applies a carefully selected set of heuristics for each of the following sub-problems:

1. **Using configurations.** We must determine whether a materialized view or a data structure of a physical design can be used to execute an operation.

2. **Determining candidate configurations.** We must determine the candidates for materialized views and physical designs.

3. **Selecting configurations.** We must determine the next configuration for the rest operations in the workload.

The main idea underlying our approach is that a solution should take operations in the workload history into account when deciding which intermediate operation results to materialize and what data structures for physical designs to create.

Algorithm 2.1 shows the overview of our solution. We keep a set of materialized views and augmented data structures for physical design in configuration $C$, and maintain the statistics $STAT$ of these data structures. When an operation arrives, we try to match it with the data structure stored in the configuration (subprocedure $matchDesign$), and we obtain a set of potential
rewritings $Rewr(Q)$. We then choose plan $Q_{\text{best}}$ which has the lowest estimated cost (subprocedure $selectRewriting$), and enumerate candidates for new materialized views and other data structures that can be generated during the execution of the operation (subprocedure $determineCandidates$). The next step is to choose the most beneficial data structures to create during the execution (subprocedure $selectDesign$). We rewrite $Q_{\text{best}}$ to $Q^{\text{instr}}_{\text{best}}$ by encapsulating the creation of the new data structures that will be admitted to the configuration (subprocedure $instrumentOperation$). The operation is then executed. The last step is to update the statistics of the new admitted data structures and the potential data structures that can be used to execute the operation (subprocedure $updateStatistics$).

Some of the above subprocedures such as $instrumentOperation$ and $executeOperation$ are straightforward and we omit the details in this work. We now discuss the main subprocedures, namely, $Match Design$, $Select Rewriting$, $Determine Candidates$, $Select Design$, and $Update Statistics$.

### 2.6.1 Match Design

When an operation $Q$ is exposed to the system, the system starts the design process by checking if $Q$ can be rewritten by using the materialized views and other augmented data structures kept in the configuration to improve the performance of $Q$. This subprocedure solves the above sub-problem 1.

Matching a design is important in our solution because executing an operation equivalent to $Q^2$ by using the data structures in the configuration may speed up the execution. We improve the workload performance through this subprocedure. Matching a design with an operation is typically a hard problem. For instance, it has been shown that matching a query with a set of materialized views is intractable in general [8]. In this thesis, in order to make our solution scalable, we utilize carefully chosen heuristics in the subprocedure of design matching depending on the specific problem. For example, we consider only a sufficient condition when we match a query with a materialized view by comparing the semantic representations of the

---

2 An operation $Q'$ equivalent to $Q$ means executing $Q'$ and $Q$ have the same result in the same database instance.
query and the view. Although we may miss some opportunities of rewriting the query with views, these heuristics make our approach scalable and are essential for practical systems. This subprocedure return a set of potential rewritings of the original operation $Q$.

2.6.2 Select Rewriting

There can be many rewritings for one operation by using different sets of materialized views and augmented data structures in the configuration, but we can only execute one of them. We compute the cost of each of such rewritings based on the statistics that we collect during the execution of the workload and our estimations, and pick the rewriting with the cheapest cost.

2.6.3 Determine Candidates

In our framework, we choose to generate materialized views and other augmented data structures for physical design during the execution of an operation. When executing an operation, we can generate many materialized views and other augmented data structures for physical design, but not all of them are useful for the workload. In order to perform the selection of the design (introduced in Section 2.6.4), we must first enumerate the candidates of materialized views and physical designs that can be created during the execution of $Q$ (solving the above sub-problem 2). Determining candidates is a key subprocedure in our solution. In this subprocedure, in addition to traditional approaches to find candidates, we also develop strategies for the specific problem such as progressive partitioning of materialized views (Chapter 3). For instance, when we enumerate candidates for partitioned materialized views, we consider how the view has been partitioned, and how an operation accesses the view. Today’s programming paradigms such as MapReduce for shared-nothing distributed systems often generate these data structures when executing an operation for purposes such as better fault tolerance. Thus for our framework, the cost of generating these data structures can often be neglected. This is a big advantage for paradigms such as MapReduce since we do not have to spend additional time to generate these data structures. We consider this property when we determine the candidates.
CHAPTER 2. THE PROBLEM OF COMBINING PHYSICAL DESIGN AND MATERIALIZATION

2.6.4 Select Design

As we have introduced in Definition 2.3, the configuration of our framework has a size upperbound. We cannot keep unlimited amount of materialized views and other augmented data structures for physical design. This not only requires unnecessary storage space, it could also make the design matching subprocedure less efficient because we may have to compute the matching between a query and a larger amount of views and other data structures in the configuration. Considering the above constraint, it is natural to select only the design that will bring the most benefit to the workload, and materialize these design during the execution of an operation.

To select the design, we use the statistics to determine what design is expected to yield the most the performance improvement (solving the above sub-problem 3). We only keep these materialized views and other augmented data structures in the system. If a new design is estimated to bring more benefit for the workload than existing designs do in the configuration and the configuration size has already reached the upperbound, we will remove the existing designs to make room for the new design. By using the statistics that we have introduced above, the select design subprocedure makes our design adapt to the evolving workload. The factors we consider in this subprocedure include the benefit of keeping a data structure such as utilizing it to execute future operations, and the cost of keeping the data structure such as creating it and the storage space it requires.

2.6.5 Update Statistics

We must update the statistics for the data structures in the current configuration and the configuration candidates. We consider the updated statistics as an abstract of the current characteristics of the workload, and utilizing these statistics appropriately in the design selection subprocedure makes our design adapt to the evolving workload.

We maintain a set of statistics of the materialized views and the augmented data structures for physical design. These statistics are crucial when we select the design (Section 2.6.4). We carefully choose what statistics are maintained for specific problems. For instance, in
a cost-based design, it is common to use the number of accesses to compute the benefit of a materialized view ([47]). Meanwhile, the number of accesses is often adjusted according to the timestamps of the accesses (also known as a decay function), so that an access that happened long time ago does not contribute as much as a recent access when we compute the benefit for the workload. With an appropriate decay function, we can make our design adapt to the evolving characteristics of the workload. This often requires the maintenance of the timestamps of all operations. We adopt this approach in the problem of progressive partitioning materialization. However, it could be expensive in terms of both storage space and computation if the number of operations is huge, which is typical for OLTP and streaming workloads, and this could become a problem.

Online data placement design for Polystore systems (Chapter 4) has such a problem. Maintaining all timestamps for all operations becomes impractical. Instead of maintaining all timestamps, we only maintain the number of operations that have accessed a fragment, in addition to the timestamp that the fragment is lastly accessed. We develop an efficient technique to adjust the number of accesses by using only the timestamp of last access (Section 4.6.2). This greatly saves the necessary storage space for all timestamps and the computation to adjust the number of accesses based on these timestamps.

In the following chapters when we study the instances of the general problem of combining materialization and physical design to improve performance for today’s real-life workloads, we will discuss the subprocedures in Algorithm 2.1 in more detail with respect to the specific problem.

2.7 Applicability

In this chapter, we introduced a problem that is paramount for scalable data management: how to efficiently combine physical design and materialization to improve the performance of real-life workloads. Our solution adopts a cost-based online approach to address two challenges for such workloads: 1) data accesses of workloads are not uniformly distributed; and 2) access
patterns of workloads often evolves.

We now introduce a few examples to show the applicability of the above problem and how our solution can be applied in these examples.

2.7.1 Progressive Partitioning of Materialized Views

We have briefly introduced the problem of progressive partitioning of materialized views in Section 1.2. The general problem that we defined in this chapter can easily be applied to this problem by defining the physical design as the partitioning of materialized views. The general solution that we developed in Section 2.6 can also be applied directly to solve this problem with only a few adaptations such as replacing the physical design by the partitioned materialized views. We will study the details in Chapter 3.

2.7.2 Online Data Placement Design for Polystore Systems

Another example of the applicability of the general problem that we defined in this chapter is the online data placement design problem that we introduced in Section 1.3. Different from the first instance problem (progressive partitioning of materialized views) that allows read-only workloads, this problem describes a scenario in which workloads contain both read and write operations. The general problem defined in this chapter is again applicable to this problem, by defining the physical design for Polystore systems as the data placement design. The solution that we described in this chapter is also applicable for solving this problem by skipping a few components that are trivial, such as Determine Candidates, and focusing on components such as Select Design. We will introduce the details in Chapter 4.

2.7.3 Online Data Co-Location and Materialization

We present the third instance problem, online data co-location and materialization, that has been briefly introduced in Section 1.4. This is a preliminary work to offer just another example to show the broad applicability of the general problem of combining physical design and materialization. Similar to the problem in Section 2.7.1, applying the general problem defined
in this chapter to online data co-location and materialization is straightforward: we only have

to define the physical design as data co-location. The adaptation of the solution developed in
this chapter is also trivial. We will introduce the details in Chapter 5.

As we have shown above, the general problem defined in Section 2.5 and its solution (Sec-
tion 2.6) have broad applicability in physical design for scalable data management. Next we
will study several instances of the problem and show how we adapt the solution to solve these
instance problems.
Chapter 3

Progressive Partitioning of Materialized Views

3.1 Introduction

In this chapter, we study an instance of the general problem that we defined in the previous chapter (Section 2.5), i.e., when the workload $W$ is read-only. Such a workload is also called an analytical workload. Typically, it contains queries that perform selection, projection, join and aggregation (SPJA). Such a workload is very common in decision support systems (DDS) and data warehouses. Materialization as an efficient technique to improve the performance of read-only workloads has been well studied in the past, but today’s exploding big data and the underlying infrastructure that supports it calls for a re-investigation of this technique. In this chapter, we introduce a physical design technique, data partitioning, to materialization and we will show that combining these two techniques efficiently can significantly improve the performance for read-only workloads.

For modern SQL systems, techniques such as materialization and data partitioning can greatly improve performance, and these techniques have been studied independently. We are the first to study the combination of materialized view selection and range partitioning.

The major advantage of creating a partitioned view from an intermediate query result is that
future queries with selection conditions over the partition attribute can be answered efficiently by accessing a subset of the view’s fragments. However, partitioning a view increases the cost of view creation. Furthermore, new challenges arise because we have to decide when to partition a view, how to select fragment boundaries (within a partitioning), when to repartition, and what fragments to evict to save space. We address these challenges in this work.

**Online View Selection.** As we have introduced in Chapter 2, a view selection algorithm that is based on a query workload is called *adaptive*. Adaptive (or workload-aware) materialization and partitioning of views may be done at design-time or at runtime. That is, either a complete workload is given and the *view selection* algorithm determines which views to materialize and how to partition them offline, or the algorithm works in an *online* fashion making decisions based on the history of queries that have been processed so far. While online materialized view selection has been studied [73], we are the first to consider the online adaptation of partitioning choices for views. Our partitioning strategy is motivated by two important characteristics of real-life data analytic workloads: 1) data access is often not distributed uniformly over the domain of a selection attribute and 2) access patterns evolve as the interests of users change over time. We have explained our study of these two characteristics in Section 2.1.

Figure 2.1 shows the characteristic of *Non-Uniform Distribution of Access*. We note that there are ranges that are rarely queried and others that are very frequently queried. Clearly, *adaptive partitioning* can improve query performance. We use the range conditions of queries to adjust fragment (partition) boundaries with the effect that *hot spots* are covered by relatively small fragments and less frequently accessed data are covered by fewer and larger fragments. This has the advantage of focusing the effort of partitioning on the parts of the data which will give us the most benefit. Queries accessing hot spots can be answered using small fragments without touching unwanted ranges of the view. Furthermore, using this approach we avoid paying the cost of partitioning data that is accessed infrequently.

Figure 2.2 shows the characteristic of *evolving access patterns*. To accommodate this, we make decisions on how to partition a view online as queries arrive. We create a materialized view with an initial partitioning once we have determined that there is enough evidence that
the creation of the view will benefit the current workload.

We progressively refine fragment boundaries based on the selection conditions of incoming queries. Our progressive partitioning, coupled with a cost-based view and fragment eviction policy, allows us to adapt to evolving workloads. A view’s competitiveness according to this cost model is based on its observed benefits (an estimate of the runtime that would have been saved if the view were to be materialized), its creation cost (the runtime overhead of materializing and partitioning the view), and its storage size. Importantly, we apply a decay function to timeout view benefits over time. This ensures that after a shift in the workload, views that are no longer useful for the current access pattern will eventually be replaced with views that fit the new pattern.

**Partitioned Materialized View Pool Size.** Typically, the storage space allocated for materialized views is not unlimited. We analyzed a BigBench workload [42] and found that if we materialize all intermediate join results as views, the total storage required is four times the size of the BigBench base tables. Of course, for evolving workloads, the number of materialized views and fragments would continue to increase and not all views will continue to provide a benefit to queries. Jain et al. [55] show the importance of a good view selection strategy for real-life applications: the savings that can be achieved with a small materialized view pool are similar to the savings that can be achieved with a large pool size as long as a good view selection strategy is applied. This implies that spending more on storage blindly for materialized views does not always bring benefits: the increased number of materialized views cannot necessarily be reused to answer queries, at the same time they can increase the cost of matching a query with the views. An important benefit of partitioned views is the finer granularity of control on view and partition selection: we can individually evict the fragments of a partitioned view that are unlikely to be used in the future.

**Correlated Fragments.** Given a finite amount of space for storing views, we present a novel strategy for selecting what fragments of a view to keep. Typically, decisions on whether to keep or evict a view are made independently for each view [47]. However, the benefits that different fragments of a partitioned view provide to a workload are not independent of each
other. Returning to Figure 2.1, observe that ranges which are accessed often (ranges with many hits) tend to have neighbors with many hits (which are also accessed often). We find similar patterns for other attributes of different SDSS tables: parts of the domain of an attribute that are close to hot spots have a higher chance of being hit in the future than parts that are further away from hot spots. We present a new probabilistic model based on this correlation to determine when a fragment of view should be evicted. Our model treats a hit to a fragment as a sample from a probability distribution. We determine the normal distribution that has the maximum likelihood to have produced the sample and use this distribution for fragment selection.

**Overlapping Fragments.** Figure 2.2 also hints at a common pattern for selection ranges. A partition containing a few large fragments (for cold spots) and several small fragments (located at hot spots) may work well for some time, but as the workload evolves, there is a need to split a large fragment as additional queries begin to access it. This split incurs high write cost for repartitioning because if a fragment is split, its whole content needs to be read and written to disk. We present a solution that permits overlapping fragments. Rather than reading and writing the large fragment, we create a small fragment that overlaps the large fragment.

**Contributions.** Our main contributions are as follows.

- **Progressive, adaptive partitioning of materialized views.** We propose the first algorithm for progressively partitioning materialized views that adapts online to changes in a query workload.
- **Exploitation of fragment correlations.** Based on our study of real-life workloads, we present a novel cost-benefit model for view fragments and candidate selection that takes the correlation among fragments of a partition into account.
- **Overlapping fragments.** We allow overlapping fragments and show that they can reduce the cost of view creation especially over evolving workloads.
- **DeepSea.** We present DeepSea, an implementation of our techniques in Hive [82].
- **Evaluation.** We demonstrate DeepSea’s effectiveness using a query workload modelled after a real workload from SDSS [5] and workloads from BigBench [42].

The remainder of the chapter is organized as follows. We discuss related work in Section 3.2, introduce preliminaries in Section 3.3, and formally state the problem addressed in
this work in Section 3.4. We give an overview of our approach in Section 3.5. We then present how to select view candidates in Section 3.6, how to select what to materialize and how to partition in Section 3.7, and how to answer queries using partitioned materialized views in Section 3.8. Afterward, we discuss the implementation of DeepSea in Section 3.9 and present our experimental evaluation in Section 3.10.

### 3.2 Related Work

There are several lines of work related to our approach: answering queries using views; reusing intermediate query results; (online) self-tuning techniques for physical database design; database cracking; and semantic caching. We have discussed the first three lines of work in Section 2.3. In this section, we discuss database cracking and semantic caching.

**Database cracking.** Database cracking [52], i.e., adaptive and progressive indexing, incrementally builds an index structure over a table based on access patterns of queries. There is a rich body of work on enhancements of cracking such as the study of robustness and adaptiveness to dynamic workloads [50]. A similarity between cracking and our approach is that they both incrementally refine physical designs based on selection conditions in queries. In contrast to cracking, DeepSea focuses on horizontal partitioning of materialized views and makes cost-based decisions on whether to refine a partition.

**Semantic caching.** Semantic caching [31] studies how to reuse subsets of input tables that are stored in a client-side cache. Each entry in the cache is described as a logical constraint (selection condition) providing a semantic description of the content of a cache entry. When a query is submitted to the client and can be answered (partially) using the cache, only a “remainder query” will be sent to the server to fetch the results that do not exist in the client’s cache. Similar to DeepSea, intermediate results are reused and are reorganized based on access patterns. However, semantic caching only considers caching of the results of selections over base tables (we consider caching of a view that is partitioned on a selection attribute) and does not allow cached regions to overlap.
3.3 Preliminaries

We have introduced a set of preliminaries in Section 2.4 for the general problem of combining materialization and physical design to improve workload performance. In this section we will introduce additional concepts and definitions that are necessary for the instance of the general problem that we study, i.e., progressive partitioning of materialization.

We first review the concept of horizontal partitioning. We use \( R_1, R_2, \ldots \) to denote relations, \( A_1, A_2, \ldots \) to denote attributes, and \( \mathcal{D}(A) \) to denote the domain of attribute \( A \). We call an attribute \( A \) ordered if there exists a total order \( \leq_A \) over \( \mathcal{D}(A) \). Only ordered attributes are considered as keys for horizontal partitioning.

**Horizontal Partitioning.** Horizontal partitioning splits the tuples of a relation into a set of disjoint fragments - each fragment holds the data for a range of values of the partition key (the attribute on which we partition). The union of these fragments equals the original relation.

**Definition 3.1 (Horizontal Partitioning).** Let \( R \) be a relation and \( A \) an ordered attribute from \( R \)’s schema. Consider a set \( \mathcal{I} = \{ I_1, \ldots, I_n \} \) of intervals where \( I_i \subseteq \mathcal{D}(A) \). The fragmentation \( \mathbb{P}_\mathcal{I}(R.A) \) of \( R \) on \( A \) according to \( \mathcal{I} \) is the set of fragments \( F_i \subseteq R \) defined as \( F_i = \{ t \mid t \in R \land t.A \in I_i \} \). If \( \bigcup_{I \in \mathcal{I}} I = \mathcal{D}(A) \) and \( \forall i, j : I_i \cap I_j = \emptyset \) then \( \mathbb{P}_\mathcal{I}(R.A) \) is called a horizontal partition.

![Horizontal Partitioning Diagram]

**Example 3.1.** Assume a relation \( R \) has 6 tuples \( \{ t_1, t_2, t_3, t_4, t_5, t_6 \} \) where the value of attribute \( A \) for tuple \( t_i \) is \( i \). The domain \( \mathcal{D}(A) \) of \( A \) is \( \{1, \ldots, 6\} \). Consider a set \( \mathcal{I} \) of three intervals \( I_1 = [1, 2], I_2 = [3, 4], \) and \( I_3 = [5, 6] \) as shown above. A partitioning based on these intervals
would result in fragments $F_1 = \{t_1, t_2\}$, $F_2 = \{t_3, t_4\}$, and $F_3 = \{t_5, t_6\}$. The fragmentation $P_I(R.A)$ is a horizontal partition of $R$ according to $A$. Consider a second set of intervals $I'$ containing $I_4 = [1, 4]$, $I_5 = [3, 4]$, and $I_6 = [5, 6]$. The fragmentation according to $I'$ results in fragments $F_4 = \{t_1, t_2, t_3\}$, $F_5 = \{t_3, t_4\}$, and $F_6 = \{t_5, t_6\}$. This fragmentation $P_{I'}(R.A)$ is not a horizontal partition of $R$, because of the overlap between $I_4$ and $I_5$. Finally, $I'' = \{I_4, I_6\}$ is again a horizontal partition of $R$.

**Overlapping Partitioning.** It is sometimes beneficial to relax the disjointness requirement by allowing fragments to overlap. We call such a fragmentation an overlapping partitioning.

**Definition 3.2 (Overlapping Partitioning).** Let $R$ be a relation and $A$ one of its attributes. We call a fragmentation $P_I(R.A)$ an overlapping partitioning iff $\bigcup_{I \in I} I = D(A)$.

**Example 3.2.** Figure 3.1 illustrates why it may be beneficial to allow fragments to overlap. Assume that a query $Q_1$ accesses a range $[a, b]$ and that based on this access pattern we have decided to create a partition with three fragments $[l, a)$, $[a, b]$, and $(b, u]$. A subsequent query $Q_2$ accesses data in the range $[a', b']$. Note that $b$ and $b'$ are close to each other. Adaptive horizontal partitioning may create four new fragments based on $Q_2$ by splitting the previously created fragments into $[a, a')$, $[a', b)$, $(b, b']$ and $(b', u]$. If we allow fragments to overlap, then we can avoid creating the fragment $(b', u]$ because no query has accessed data from this fragment yet. Instead, we create a fragment $(b, b']$ and keep the fragment $(b, u]$ that was created based on $Q_1$. This avoids writing a large fragment that may not be accessed by future queries at the cost of additional storage for $(b, b']$.

We adapt the definitions for workload (Definition 2.1) and workload history (Definition 2.2) by requiring that all queries in workload $\mathcal{W}$ are read-only. The workload history is an indicator of the characteristics of the workload, and our online design approach is based on the workload history.
3.4 Problem Statement

We now state the problem addressed in this chapter: how to maintain a set of partitioned views (the *materialized view pool*) in an online fashion in order to maximize query performance.

First we adapt the definition for configuration (Definition 2.3) by introducing partitioned views as the data structure of physical design.

**Configuration.** A configuration $C$ models the current content of the materialized view pool. It consists of the set of views $V$ that are currently in the pool and a mapping $P$ that associates each view $V$ and one of its attributes $A$ with a set of intervals describing the current partitioning of the view on this particular attribute. Note that we permit multiple partitions of a view to be stored in the pool as long as these partitions are on different attributes. We define $P(V, A) = \emptyset$ if view $V$ has not been partitioned on attribute $A$ yet.

**Definition 3.3.** A configuration $C$ is a pair $(V, P)$ where $V$ is the set of views materialized in the pool and $P$ is a mapping that associates with each view $V \in V$ and an attribute $A$ in the schema of $V$ a set of intervals $I$ over the domain $D(A)$ of $A$.

Note that there is an upperbound $S_{\text{max}}$ for the size of a configuration. We use $S(C)$ to denote the total storage size of the views in configuration $C$, and $S(C) \leq S_{\text{max}}$.

**Problem Definition.** Now we adapt the problem definition in Section 2.5 to progressive partitioning materialization. In this work, we assume a read-only workload, i.e., no updates. We address the following problem: given a workload $\mathcal{W} = Q_1, \ldots, Q_n$ of queries to be executed
that is unveiled one query at a time, choose a sequence of configurations \( C = C_1, \ldots, C_n \) in order to minimize the total execution time of the workload plus the time spent on view creation

\[
\text{COST}(\mathcal{W}, C) = \sum_{i=1}^{n} \text{COST}_{\text{EXE}}(Q_i, C_i) + \sum_{i=1}^{n-1} \text{COST}(C_i, C_{i+1}).
\]

Here \( \text{COST}_{\text{EXE}}(Q, C) \) denotes the cost of executing query \( Q \) given configuration \( C \) and \( \text{COST}(C_i, C_{i+1}) \) denotes the cost of creating configuration \( C_{i+1} \) from configuration \( C_i \). We require \( C_1 = \emptyset \), i.e., no views have been created before the workload execution. We are interested in a restricted version of this problem where new views and refinements of partitions have to be based on a query plan \( p_{ij} \) of the currently executed query \( Q_i \), i.e., only views and fragments corresponding to intermediate results of this query (\( V_{\text{cand}}(p_{ij}) \) and \( P_{\text{cand}}(p_{ij}) \), defined in Section 3.6) are considered as candidates to be added to \( C_{i+1} \). Given these preliminaries we can state the online partitioned view selection problem as follows.

**Definition 3.4** (Online Partitioned View Selection). Given workload \( \mathcal{W} = Q_1, \ldots, Q_n \) that is unveiled one query at a time, incrementally determine the sequence of configurations \( C = C_1, \ldots, C_n \) that minimizes

\[
\text{COST}(\mathcal{W}, C) = \sum_{i=1}^{n} \text{COST}_{\text{EXE}}(Q_i, C_i) + \sum_{i=1}^{n-1} \text{COST}(C_i, C_{i+1})
\]

subject to

1. \( C_1 = \emptyset \)
2. \( C_{i+1} - C_i \subseteq V_{\text{cand}}(p_{ij}) \cup P_{\text{cand}}(p_{ij}) \) for all \( i \in \{1, \ldots, n-1\} \)

The online partitioned view selection problem is difficult for several reasons. First, this is an online problem: for each incoming query \( Q_i \), we must decide which partitioned views or fragments to create and which to evict from the pool without knowing the remaining sequence of queries from the workload. There is abundant literature for online algorithms that provide worst-case guarantees. An online algorithm is said to be \( k \)-competitive if its result is at most of a factor \( k \) worse than the solution computed by an optimal offline algorithm (an algorithm which has access to the whole input). However, the competitiveness factor of such algorithms for search space sizes encountered in our problem are too high to be of any practical relevance.
Even if we were to consider the offline version of the problem, we cannot hope for an optimal solution because of the undecidability of query answering with views.

Given these constraints we strive for a principled yet scalable solution that applies a carefully selected set of heuristics for each of the sub-problems of determining view and partition candidates, view and partition selection (determine the next configuration), and view and partition matching (determining whether a partitioned view can be used to answer a query). The main idea underlying our approach is that a solution should take hints provided by queries in the workload into account when deciding which intermediate query results to materialize and how to partition them.

### 3.5 Solution Overview

We adapt Algorithm 2.1 in Section 2.6 to solve the problem defined by Definition 4.8. Algorithm 3.1 gives a high-level view of the approach we use to process a query. The main difference is that we define the detail data structures that are kept in the configuration in Algorithm 3.1, i.e., $V_{\text{cand}}$ in Algorithm 2.1 becomes $(V_{\text{cand}}, P_{\text{cand}})$, and $V_{\text{sel}}$ becomes $(V_{\text{sel}}, P_{\text{sel}})$.

In the first step we determine which views and fragments (in the configuration or not) can be used to answer the query (Section 3.8). The result of this step is a set $\text{Rewr}(Q)$ of possible rewritings of the input query which use the views. We update the statistics kept for partitioned views to record that some views/fragments can be used to answer the query, rather than updating them at the end of the solution as Algorithm 2.1 because we have enough information about these potential views/fragments. Afterwards, among the rewritings that only use queries which are currently in configuration $C$ we determine the rewriting $Q_{\text{best}}$ with the lowest expected cost. Now that we have chosen a “plan” for the query (Section 3.6), we determine which of the intermediate results of the query are viable candidates to be stored as materialized views $(V_{\text{cand}})$ and how to partition them $(P_{\text{cand}})$. Note that even if a view $V \in V_{\text{cand}}$ already exists and is partitioned, we may still produce fragment candidates for it (e.g., splitting an existing fragment to create a refined partition). Given such sets of candidates we add them to the set
of partitioned views for which we want to keep statistics (using an initial rough estimate of
their costs and benefits). The next step, described in more detail in Section 3.7, is to determine
which of these candidates should be materialized during the execution of $Q_{\text{best}}$ and, if neces-
sary, which views to evict from the current configuration $C$ to make space for these new views
(recall that we limit the configuration size by $S_{\text{max}}$). Once we have selected the views $V_{\text{sel}}$ and
fragments $P_{\text{sel}}$ to create, we instrument the query $Q_{\text{best}}$ to materialize intermediate results (and
partition them if need be). We then execute the instrumented query $Q_{\text{instr}}^{\text{best}}$ and return its result
to the user. Finally, we update the statistics for all candidates based on the information gained
by executing $Q_{\text{best}}$, e.g., we now have precise measurements for the size of candidate views.

### 3.6 View and Partition Candidates

We now discuss how our approach determines which views and fragments to create for a given
query $Q$ and configuration $C$. Our creation process operates in two steps: first we determine
for which views and fragments we have gathered enough evidence to materialize them and then
based on this subset of candidates we determine the next configuration based on the “value” of
a view or a fragment using the statistics that we keep.

**View and Fragment Statistics.** For each view or fragment candidate, no matter whether
materialized in the pool or not, we store statistics such as its size $S$, the estimated cost of
creating it ($\text{COST}$), the set of timestamps when this view could have been used to answer a
query ($T$), and a list of potential savings associated with each such timestamp ($B$). $B$ and $T
together with a decay function that times out benefit as mentioned in Section 3.1, are used to compute the benefit of a view. For fragments we only record $T$ and $S$ since the benefit can be inferred based on its size and the saving of the view this fragment belongs to. Similarly, $\text{COST}$ of a fragment is determined based on $\text{COST}$ for its view.

**Definition 3.5.** The view statistics $\text{STAT}$ is a triple $(V_{\text{STAT}}, P_{\text{STAT}}, \Sigma)$ where $V_{\text{STAT}}$ is a set of views, $P_{\text{STAT}}$ is a mapping as in $C$ that associates each view and attribute in its schema with a set of fragment intervals, and $\Sigma$ maps each view in $V_{\text{STAT}}$ and fragment in $P_{\text{STAT}}$ to a tuple $(S, \text{COST}, T, B)$ respective $(S, T)$.

### 3.6.1 View Candidates

We first notice that certain relational operators are less likely to provide results that can be reused or the reuse of such an operator’s result would not result in significant performance improvement. We consider the intermediate results of the following operators as candidates: join, aggregation, and projection. Joins are good candidates, because join computation is expensive and join results are likely to be reused. We consider aggregation operators, because the result size of an aggregation is typically small while its input size is large. Thus, we can save large computational cost by paying a small storage and creation cost. Likewise, projections can also reduce the size of their input considerably. We do not consider selections as view candidates, because materializing the input of the selection and partitioning it on the attribute used in the selection is usually more effective than using selections alone.

In this work, we only consider query plans started with join, aggregation and projection.

**Definition 3.6 (View Candidates).** For a query $Q$ and view configuration $C$, the set $\mathcal{V}_{\text{cand}}(Q)$ of view candidates for $Q$ contains all subqueries $Q'$ of $Q$ that fulfill the following conditions:

- $Q'$ is of the form $\gamma(Q_1)$, $Q_1 \bowtie Q_2$, or $\pi(Q_1)$
- $Q'$ does not exist in $\mathcal{V}$
3.6.2 Partition Candidates

Similar to our view candidate generation approach, we want to use the characteristics of the current workload to guide the partition candidate generation. Note that we may maintain multiple partitions of the same view on different attributes. Given a current configuration of partitioned views $C$ and statistics $\text{STAT}$ kept for this configuration as well as for candidates, we consider new fragment candidates based on the selection conditions applied by a query. For every conjunction in the condition of a selection, i.e., a selection $\sigma_{l \leq A \leq u}(Q')$, which is a subquery of the current query $Q$, we consider new partition candidates for the view corresponding to $Q'$, say $V$, based on the selection condition over attribute $A$. For the following discussions, without loss of generality, we assume $l \geq \bar{A}$ where $\bar{A}$ is the lowerbound of the domain of $A$, and $u \leq \bar{A}$ where $\bar{A}$ is the upperbound of the domain of $A$. It is trivial to replace $l$ with $\bar{A}$ and similar for $u$ when the above conditions do not hold. We have to distinguish several cases: 1) if we have not materialized $Q'$ as a view $V$ yet. In this case, we use $l$ and $u$ to split the potential fragments in $P_{\text{STAT}}(V, A)$ which contain these points. If we have not yet gathered any intervals for this partition of $V$ yet ($P_{\text{STAT}}(V, A) = \emptyset$), then we initialize the partition with a single fragment: $\{D(V, A)\}$ and then use $l$ and $u$ to split this fragment; 2) if a view $V$ corresponding to $Q'$ and a partition $P(V, A)$ on attribute $A$ already exists, then we again use the end points of the interval defined by the selection condition to consider splits of existing fragments that contain an end point as candidates. For each interval $I' = [l', u']$ from $P(V, A)$ and the interval $I = [l, u]$, we create new candidates if either $l \in I'$ or $u \in I'$ using $l$ respective $u$ (or both) as split point(s).

**Definition 3.7 (Partition Candidates).** Let $Q$ be a query, $C$ a view configuration, and $\text{STAT}$ a view statistics. Consider a subquery $\sigma_{l \leq A \leq u}(Q')$ of $Q$ where $Q'$ corresponds to a view $V$ in $V_{\text{STAT}}$ and the intervals associated with partitioning $V$ on attribute $A$ (either $P(V, A)$ if the view is in the pool or $P_{\text{STAT}}(V, A)$ otherwise). We use $I$ to denote $[l, u]$. For every interval $I' = [l', u']$ from $P(V, A)$ respective $P_{\text{STAT}}(V, A)$ we define the set of partition candidates $P_{\text{cand}}(V, A, Q')$ according to $V$, $Q$, and $Q'$ as the union of the sets of candidates for every such $I'$.
1. There is no overlap between these two intervals, i.e., \( I' \cap I = \emptyset \). In this case, no candidates are generated.

2. The query selection interval contains the partition interval, i.e., \( I' \subseteq I \). In this case, no candidates are generated.

3. The query selection interval overlaps the fragment interval from the left, i.e., \( l < l' < u < u' \). In this case, intervals \([l', u]\) and \((u, u']\) are considered as candidates.

4. The query selection interval overlaps the fragment interval from the right, i.e., \( l' < l < u < u' \). In this case, intervals \([l', l)\) and \([l, u']\) are considered as candidates.

5. The query selection interval is contained in the fragment interval, i.e., \( I \subset I' \). In this case, we consider three intervals as candidates: \([l', l)\), \([l, u]\), and \((u, u']\).

**Example 3.3.** Consider a view \( V(A, B) \) that is partitioned on attribute \( A \) using intervals \( I_1 = [0, 10] \), \( I_2 = (10, 20] \) and \( I_3 = (20, 30] \). For an incoming query \( Q = \sigma_{5 \leq A \leq 25}(V) \) we would consider the following candidates. Interval \( I = [5, 25] \) overlaps with \( I_1 \) on the right (case 4). Thus, we create candidates \([0, 5)\) and \([5, 10]\). No candidates are generated for \( I_2 \) (case 2). Finally, \( I \) overlaps with \( I_3 \) from the left (case 3) and we generate additional candidates \((20, 25]\) and \((25, 30]\).

---

**3.7 Views and Partition Selection**

Our view and fragment selection method consists of two steps: 1) exclude candidates for which we have not gathered enough evidence of their effectiveness in improving the performance of the workload and 2) decide which candidates to materialize and which ones to evict to keep the pool size below the limit \( S_{max} \). The second step ranks views and fragments based on their...
value ($\Phi$) as defined below. For each new fragment, we either create it by splitting existing fragments or create it as an overlapping fragment.

### 3.7.1 Cost and Benefit Model

We use a heuristic cost-benefit model to keep track of the “benefits” of view and fragment candidates. The benefits of a candidate are computed based on the potential savings in query execution time if this candidate would have been used to answer queries from the workload, the cost of creating it, its storage size, and other useful statistics for views and fragments. For candidates that have not been generated yet, we estimate their storage size and creation cost. We use this information to select which candidates to materialize and to decide which candidates to evict to make space for more competitive candidates.

**View Statistics.** For each view (candidate) $V$ we keep the following statistics in $V_{\text{STAT}}$: the storage size $S(V)$ occupied by the view, a set of timestamps $T(V)$ when the view was used to answer a query, and the creation cost of the view $\text{Cost}(V)$ (which is initially estimated when we first see this view as a candidate). The creation cost is replaced with the actual cost once the first query containing the view as a subquery has been executed. The same applies to $S(V)$.

We compute the accumulated benefit $B(V, t_{\text{now}})$ for a view at time $t_{\text{now}}$ as follows. $B(V, t_{\text{now}})$ is the cost we (could) have saved by using the view. The benefit is defined as

$$B(V, t_{\text{now}}) = \sum_{Q \text{ used } V \text{ at } t} \left( \text{Cost}(Q) - \text{Cost}(Q/V) \right) \cdot \text{Dec}(t_{\text{now}}, t)$$

where $\text{Cost}(Q)$ is the cost of query $Q$ without using the view, $\text{Cost}(Q/V)$ is the cost of running the query when using view $V$, and $\text{Dec}(t_{\text{now}}, t)$ is a monotonically decreasing function (in $t_{\text{now}} - t$) mapping the current time ($t_{\text{now}}$) and time when query $Q$ was executed ($t$) to a value in $[0, 1]$. $\text{Dec}(t_{\text{now}}, t)$ is used to weight past cost savings by their age. This enables our approach to adapt to a changing workload. In our implementation we use the decay function as defined below which times out any benefit after a threshold $t_{\text{max}}$ and otherwise counts it
proportionally based on \( \frac{t}{t_{\text{now}}} \).

\[
\text{DEC}(t_{\text{now}}, t) = \begin{cases} 
0 & \text{if } (t_{\text{now}} - t) > t_{\text{max}} \\
\frac{t}{t_{\text{now}}} & \text{otherwise}
\end{cases}
\]

**View Value.** Similar to Nectar [47], for each view \( V \) in the pool and candidate in \( \mathcal{V}_{\text{STAT}} \) we compute its “value” at time \( t_{\text{now}} \) as a cost-benefit ratio \( \Phi(V, t_{\text{now}}) \). We use \( \Phi \) during view selection to determine which views should be in the next configuration (views with a higher value are preferred over views of lower value). Using \( \text{Cost}(V) \), the accumulated benefit \( B(V, t_{\text{now}}) \), and size \( S(V) \), we define \( \Phi(V, t_{\text{now}}) \) as:

\[
\Phi(V, t_{\text{now}}) = \frac{\text{Cost}(V) \cdot B(V, t_{\text{now}})}{S(V)}
\]

The intuition behind the definition of \( \Phi(V, t_{\text{now}}) \) is that when a view is expensive to generate or the accumulated benefit of the view is large, its value is high and it is preferred over views with lower value. On the other hand, if the size of the materialized view is large, it is less competitive than other views of smaller size and similar benefits.

**Fragment Statistics.** Similar to view statistics we also keep separate statistics for every fragment in \( \mathcal{P}(V, A) \) (a partition of view \( V \) on attribute \( A \) that is in the pool) as well as \( \mathcal{P}_{\text{STAT}}(V, A) \) (a potential fragment candidate which is currently not materialized, but we have considered as a candidate before). For each such interval \( I \) (corresponding to a fragment \( F \)) we maintain the following information: the **storage size** of the fragment \( S(I) \), its **creation cost** \( \text{Cost}(I) \), and a set of **timestamps** \( T(I) \) when the fragment was hit (it was or could have been used to answer a query). These timestamps are used to compute the fragment value in a similar fashion as the view value explained above. We define the cost of creating the fragment to be the same as the cost of creating the partitioned view this fragment belongs to. This is because in order to recompute the fragment if it is not in the pool, we have to recompute the view’s query and partition it.
**Fragment Value.** The value of a fragment is also modeled as a cost-benefit ratio in the same fashion as for views with the exception that benefits are computed as a ratio of the view creation cost and the relative size of the fragment compared to the total size of the view. The accumulated benefit for a fragment $I$ is computed as

$$B(I, t_{now}) = \sum_{Q \text{ used } I \text{ at } t} \left( \frac{S(I)}{S(V)} \cdot \text{Cost}(V) \cdot \text{DEC}(t_{now}, t) \right)$$

and

$$\Phi(I, t_{now}) = \frac{\text{Cost}(V) \cdot B(I, t_{now})}{S(I)}$$

**Probabilistic Fragment Benefit Model.** The definition of the value of a fragment above ignores the fact that fragments in a partition of a view do not exist independent of each other, i.e., two fragments may be “neighbors” (e.g., $[0, 10]$ and $[11, 30]$) or may be quite dissimilar (e.g., $[0, 10]$ and $[1000, 1010]$). If we treat the hits on fragments we have observed so far in the workload as samples of a probability distribution, then when using these samples to determine the underlying distribution, it would be natural to consider “distance” between fragments in the mechanism that determines the distribution. For instance, if we observe a large number of hits on a fragment $[0, 5]$ and no hits on fragments $[6, 10]$ as well as $[11, 15]$, then it is still reasonable to assume that fragment $[6, 10]$ which is close to a “hot spot” has a higher likelihood to be used in the future than fragment $[11, 15]$. Based on this observation, we present a mechanism for adjusting the number of hits per fragment. Define the number of hits $H(I)$ for a fragment $I$ as $H(I) = \sum_{Q \text{ used } I \text{ at } t} \text{DEC}(t_{now}, t)$. We now define the adjusted number of hits $H_{A}(I)$ to compute a more realistic fragment value.

Consider a partition $\mathcal{P}_{I}(V.A)$ for a view $V$. Note that we do not require that all intervals in $\mathcal{I}$ are currently in the pool. We keep statistics for each $I \in \mathcal{I}$ no matter whether materialized or not. Let $H_{total}$ denote the total hits over all fragments of $\mathcal{I}$ adjusted by our decay function, i.e., $H_{total} = \sum_{I \in \mathcal{I}} H(I)$. $H_{total}$ is the total number of queries that used at least one fragment from $\mathcal{P}_{I}(V.A)$ weighted by $\text{DEC}(t_{now}, t)$.
In the analysis of the real-life workloads presented in Section 2.1 and Section 3.1, we observe that ranges which are accessed often tend to have neighbors which are also accessed often. It is reasonable to assume that a normal distribution underlies accesses to values of an attribute’s domain. Thus, given the observed hits for fragments we want to choose the mean \(\mu\) and variance \(\sigma^2\) of a normal distribution such that the resulting distribution best fits the observed hits. Here we apply well-known techniques from statistics for computing the maximum likelihood estimators (MLE) for the mean \(\hat{\mu}\) and variance \(\hat{\sigma}^2\) of normal distributions [84] to do the curve fitting.

We split the domain of attribute \(A\) into equi-size intervals \(p_1, \ldots, p_n\) which we call parts to distinguish them from fragments. We choose a quantification such that no part \(p_i\) is partially contained in an interval \(I \in I\). For instance, for a domain \([0, 20]\) if \(I = \{[0, 10], [11, 15], [16, 20]\}\) we may choose parts of size 5: \(\{[0, 5], [6, 10], [11, 15], [16, 20]\}\). Based on the hits recorded for fragments \(I \in I\) we then determine the hits for each part \(p_i\). For each fragment, we split the number of hits to this fragment evenly to the parts that are contained in the fragment. Let \(I' \subseteq I\) be the intervals containing \(p_i\) and \(#I\) the number of parts contained in interval \(I\). We define \(H(p_i) = \sum_{I \in I'} \frac{H(I)}{\#I}\), i.e., summing up the number of hits for each interval containing the part weighted based on the number of parts the interval contains. The likelihood function \(L\) for a standard distribution \(N(\mu, \sigma)\) and set of observations \(\{p_1, \ldots, p_n\}\) determines how likely it is that this particular distribution produced the given set of observations. It is defined as:

\[
L(\mu, \sigma^2; p_1, p_2, \ldots, p_n) = (2\pi\sigma^2)^{-n/2} \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^{n} (p_i - \mu)^2\right)
\]

By solving the log-likelihood function of the above function we have the maximum likelihood estimator mean and variance:

\[
\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} p_i \quad \quad \quad \hat{\sigma}_n^2 = \frac{1}{n-1} \sum_{i=1}^{n} (p_i - \hat{\mu}_n)^2
\]

The distribution \(N(\hat{\mu}, \hat{\sigma})\) is the normal distribution which is most likely given the observations.
(it maximizes the likelihood function \( L \)). Note that we use the adjusted sample variance for the estimator \( \hat{\sigma}^2 \) because usually we do not expect a very large number of fragments for a view. This is a standard approach in statistics [84].

Note that since the MLE method is inexpensive we repeatedly adapt the estimation during the selection process for each incoming query. Based on the smoothed distribution of value accesses \( N(\hat{\mu}, \hat{\sigma}) \) we get from the maximum likelihood method and \( H_{total} \), the total number of hits over all partitions, we compute the adjusted hits for a fragment \( I = [l, u] \) as:

\[
H_A(I) = H_{total} \cdot (P(x \leq u) - P(x \leq l))
\]

Here \( P(x \leq c) \) is an estimate (which ignores interval overlap) of how likely an access to a point in the interval \([-\infty, c]\) is computed over the normal distribution we have estimated using MLE. Note that this technique works for any probability distribution such as a Zipfian distribution or a mixture of distributions as long as it is feasible to compute the MLE of such a distribution given the observations. Here we choose the normal distribution, because it closely resembles the access patterns we have found in the real world workloads we have studied.

### 3.7.2 Filtering View and Partition Candidates

Our goal is to only save an intermediate result as a materialized view if this view is likely to be reused in the future and if the benefit of reuse \( B(V, t_{now}) \) will offset the cost \( \text{COST}(V) \) of materializing this view. Thus, the subset of candidates we consider for materialization is:

\[
V_{sel} = \{ V \mid V \in V_{cand} \land \text{COST}(V) \leq B(V, t_{now}) \}
\]

We apply a similar filtering step for fragment candidates. This step is only applied for fragment candidates of existing partitions, i.e., when we decide whether to refine an existing partition based on selection, but not for candidates fragments for partitions which are not in the pool yet. Here we use the total benefits for a fragment computed based on its adjusted hits (using the estimated probability distribution of hits). Consider a candidate fragment \( I_{cand} \) for
partition $\mathcal{P}(V, A)$ that is a candidate for the current query. The cost of creating $I_{\text{cand}}$ depends on which fragments are currently in $\mathcal{P}(V, A)$. To materialize $I_{\text{cand}}$ we have to read all fragments $I$ such that $I \cap I_{\text{cand}} \neq \emptyset$, extract data that belongs to $I_{\text{cand}}$ and then store $I_{\text{cand}}$. While we do not know upfront the actual size $S(I_{\text{cand}})$ for a fragment $I_{\text{cand}}$, we can estimate it based on the sizes of fragments currently in $\mathcal{P}(V, A)$ that overlap with $I_{\text{cand}}$. We assume that values are uniformly distributed within each fragment, and thus we can use the relative overlap between $I_{\text{cand}}$ and an intervals in $\mathcal{P}(V, A)$ to estimate the size as:

$$S(I_{\text{cand}}) = \sum_{I \in \mathcal{P}(V, A): I \cap I_{\text{cand}} \neq \emptyset} \frac{\|I_{\text{cand}} \cap I\|}{\|I\|} \cdot S(I)$$

Based on this estimate of the size for a candidate fragment we have not materialized yet (otherwise we would know its size) we estimate the cost of creating the fragment as:

$$\text{Cost}(I_{\text{cand}}) = w_{\text{write}} \cdot S(I_{\text{cand}}) + \sum_{I \in \mathcal{P}(V, A): I \cap I_{\text{cand}} \neq \emptyset} w_{\text{read}} \cdot S(I)$$

Here $w_{\text{read}}$ (and $w_{\text{write}}$) denote implementation specific constants for reading (respectively, writing) data. In our implementation of DeepSea, $w_{\text{write}}$ is typically much larger than $w_{\text{read}}$ if we store a fragment in HDFS. Given the cost and estimated size, we only consider fragments for which the benefits are larger than the creation cost:

$$\mathcal{P}_{\text{sel}} = \{ I \mid I \in \mathcal{P}_{\text{cand}} \land \text{Cost}(I) \leq B(I) \}$$

### 3.7.3 View and Fragment Selection

Given the prefiltred set of candidate views $\mathcal{V}_{\text{sel}}$, we now determine which of them to materialize (admit to the pool). In case this causes the total size of the views and fragments to exceed the limit $S_{\text{max}}$, we also have to decide which views or fragments to evict from the pool. Note that for selection we treat each fragment of a view independently. That is, the views in the pool do not partake in the selection process, only their fragments. However, candidate views
and fragments are treated alike (candidate fragments are only created for partitioned views in the pool and view candidates are only created for views that do not currently exist in the pool). Thus, the set of views and fragments that are considered to be selected for the next configuration are:

\[
\text{ALLCAND} = \mathcal{V}_{sel} \cup \mathcal{P}_{sel} \cup \bigcup_{V \in \mathcal{V}, A \in \text{SCHEMA}(V)} \mathcal{P}(V, A)
\]

We rank the elements (views and fragments) in this set based on their value \(\Phi\) (defined in Section 3.7.1). We then greedily add elements to the new configuration based on their rank.

Let \(\text{ALLCAND}[i]\) be the \(i^{th}\) element from \(\text{ALLCAND}\) according to \(\Phi\) in decreasing order. We keep the first \(n\) elements from \(\text{ALLCAND}\) for the largest \(n\) such that \(\sum_{i=0}^{n} S(\text{ALLCAND}[i]) \leq S_{\text{max}}\):

\[
C_{i+1} = \{\text{ALLCAND}[i] \mid i \in \{0, \ldots, n\}\}
\]

where

\[
n = \arg\max_{j \in \mathbb{N}} \left(\sum_{i=0}^{j} S(\text{ALLCAND}[i]) \leq S_{\text{max}}\right)
\]

### 3.8 View and Partition Matching

The first important step when processing a query \(Q\) with our approach is to determine which views (whether in the pool or not) can be used to answer query \(Q\). We call this process matching. The purpose of this step is to 1) update the statistics of views and fragments that could be used to answer query \(Q\) and 2) to determine the most efficient way of executing the query given the current configuration. The problem of finding all rewritings of a query \(Q\) given a set of views, i.e., queries that use the views and are equivalent to the input query, has often been called query answering with views. As mentioned earlier this problem in its full generality is undecidable for the class of queries we are interested in. We adopt a technique from Goldstein
and Larson [46] that uses a sufficient condition to determine whether a view can be used to answer a query and indexes views such that this condition can be efficiently tested. We use a modified version of the index structure introduced in this work adapted for partitioned views to speed up matching.

3.8.1 A Sufficient Condition for Matching

The sufficient matching condition of Goldstein and Larson is checked over a representation of the query and the view (called signature) which is mostly independent of syntax, but can nonetheless be constructed from a concrete plan for the query. Signatures abstract away certain syntactic features such as join order. Our logical matching approach compares subqueries of a query with materialized views by computing the signatures for both the view and the subquery, and then checking the sufficient condition of Goldstein and Larson. Thus, we are able to also match parts of a query with a view. The signature of a query consists of the set of relations accessed by the query (relation classes), information on join and selection predicates (attribute equivalence classes, selection predicate ranges, and remaining selection predicates), projections, aggregation functions and group-by expressions. We refer the reader to Goldstein and Larson [46] for definitions of these abstractions.

3.8.2 Partition Matching

Once we have determined a rewriting using the views, the next step is to determine which partition of each view included in the rewriting to use and for each partition determine a subset of the fragments to be used. In order to match a fragment and a query, we must first find a match between the view represented by the fragment and the query. Note that a fragment of a view $V$ corresponds to a view $\sigma_{l<A<u}(V)$ where $A$ is the attribute on which $V$ is partitioned on, and $u$ and $l$ are the boundaries of the fragment.

For every view $V$ partitioned on $A$ that is matched against a subquery $Q'$ of the current query $Q$, we determine the restrictions $Q'$ places on attribute $A$. This is done by using information about value ranges of selection conditions that are stored in the Attribute Value Ranges part of
Algorithm 3.2 Partition Matching Algorithm

1: procedure PARTITIONMATCHING($\theta, I$)
2: \( u_\theta \leftarrow \text{Upperbound of } \theta \)
3: \( l_\theta \leftarrow \text{Lowerbound of } \theta \)
4: \( F \leftarrow \emptyset \)
5: \( u_{\text{covered}} \leftarrow l_\theta \)
6: while \( u_{\text{covered}} < u_\theta \) do
7: \( I_{\text{cand}} \leftarrow \{ I \mid I \in I \land l \leq u_{\text{covered}} \land l > l_{\text{covered}} \} \)
8: \( I_{\text{cur}} = \text{argmax}_{I \in I_{\text{cand}}} \frac{I}{l} \)
9: \( u_{\text{covered}} \leftarrow I_{\text{cur}} \)
10: \( F \leftarrow F \cup \{ I_{\text{cur}} \} \)
11: end while
12: return \( F \)

For a query’s signature (see [46] for a detailed explanation of the signature). Given our definitions of overlapping partitioning, the matching between a set of overlapped fragments and a query selection range is a set cover problem and thus is intractable. We use Algorithm 3.2 that greedily matches the fragments to a query selection range. Note that we use \( I \) to denote the lower and \( \bar{I} \) to denote the upper bound of an interval \( I \). We look for a set of fragments whose union covers the selection range. We maintain a variable \( u_{\text{covered}} \) that stores the upper bound of the region covered so far. \( u_{\text{covered}} \) is initialized to the lower bound of the selection range of the query \( u_\theta \). In each iteration of the loop, we greedily add the fragment that has the largest lower bound among the fragments that cover \( u_{\text{covered}} \) from the left.

### 3.8.3 Indexing Partitioned Views

When computing matches between a query \( Q \) and a set of materialized views, it would be too slow to evaluate the sufficient matching condition over the signatures of all pairs of subqueries of \( Q \) and views in the pool. We adapt an in-memory index for view signatures called a filter tree [46] to be able to prune the search space early-on. A node in the tree is represented by a set of (key, pointer) pairs, where the key is a set of values, and the pointer points to a node on the next level. Each level represents one of the signature parts, e.g., the relations accessed by the view. The pointer of a leaf node points to a view. For each view, we store its partition information. For each partition of a view, we store the boundaries and statistics for each of its fragments. Note that we allow multiple partitions for the same view to exist as long as they are on different attributes. The search key for a query \( Q \) is its signature. We also use this index to
keep the statistics for view and partition candidates (covered in Section 3.6).

3.8.4 Updating View and Partition Statistics

During view matching we update the statistics we keep for each view and its fragments, no matter whether the view or fragment is currently in the pool or not. For every rewriting $Q_{rewr} \in \text{Rewr}(Q)$ let $V$ be a view that has been used in $Q_{rewr}$ and for each such view $\mathcal{P}(Q_{rewr}, V, A)$ be the fragments of the partition of $V$ on attribute $A$ that are accessed by $Q_{rewr}$. We update the statistics for each such view and its fragments to reflect that it could be used to answer the query $Q$ using the formulas presented in Section 3.7.1.

3.9 Implementation

DeepSea extends Hive (version 0.8.1) [82], an SQL-on-Hadoop system [1]. While we have chosen Hive, because it is relatively mature, our techniques are applicable for any system that supports declarative querying on-top of shared-nothing dataflow systems.

**Query Processing in DeepSea.** Figure 3.2 shows how a query is processed by DeepSea. We use the parser and semantic analyzer of Hive to transform the input query into an abstract syntax tree (AST). The AST is translated into a directed acyclic graph (DAG) of operators (operator tree) and a task DAG (task tree) is generated from the operator DAG. Task DAGs assign operators to map and reduce phases. Our view matching module (see Section 3.8) rewrites the task DAG by replacing subqueries with references to materialized views or fragments. The rewritten DAG is then transformed into a DAG of MapReduce jobs to form a execution plan. We have implemented a partition operator that splits its input based on a list of fragment predicates which determine which input tuple belongs to which fragment. The output for each fragment is routed to a file sink operator that writes the fragment’s content to a file.

**Simulator.** Testing view and fragment selection strategies requires extensive experiments over a large number of diverse workloads. Since the benefits of partitioned views are more pronounced for large datasets, it is necessary to consider such datasets which results in large query
runtimes. To be able to quickly test variations of a workload with different selection conditions ranges we have developed a simulator to study the efficiency of our selection algorithm and compare it to alternative approaches. We run a series of query templates with different selection patterns (introduced in Section 3.10) and gather statistics such as the storage size of views and fragments as well as the elapsed time. The simulator keeps track of the query template and the selection pattern that is running. It builds the necessary views and partitions based on the selection strategies and the size limit of the materialized view pool. Once sufficient statistics have been gathered for a query template, we estimate the runtime of future executions of a query template using linear regression.

**Bounding Fragment Size.** There are situations where bounding the size of a fragment (from above or below) may be beneficial. If the access patterns of queries are limited to a small subrange of the domain of an attribute, then our approach may create very large fragments for the parts of the domain that are accessed infrequently. In general it would be beneficial to split such large fragments, because the potential benefit of large fragments is small while the overhead of creating a few medium sized fragments instead is not very high. We approach this problem by limiting the maximal size of the fragments we create relative to the size of a view. We define a threshold $\phi$ for the relative size of a fragment. When we materialize and partition a view, we split every fragment that is larger than $\phi \times S(V)$ into smaller, equi-sized fragments. Big data systems are usually built on top of distributed file systems that favors large block sizes. For instance, HDFS has a default block size of 128 MB (or 64 MB depending on the version). We use the file system’s block size as the lower bound for fragment size.
### Table 3.1: Parameters and their values

<table>
<thead>
<tr>
<th>Description</th>
<th>Values (default in bold)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance size</td>
<td>100GB, 500GB</td>
</tr>
<tr>
<td>Pool size</td>
<td>50GB, 125GB, 250GB, 500GB, ∞</td>
</tr>
<tr>
<td>Query selectivity</td>
<td>1% (Small), 5% (Medium), 25% (Big)</td>
</tr>
<tr>
<td>Query skew</td>
<td>Uniform (U), Light (L), Heavy (H)</td>
</tr>
</tbody>
</table>

3.10 Evaluation

We evaluate our system using the big data benchmark suite BigBench [42]. We demonstrate the overall performance of DeepSea using queries and data distributions that are modeled based on the SDSS workload [5]. This ensures that our evaluation considers important characteristics of real workloads. We also use BigBench to generate a set of synthetic workloads that are tailored to evaluate our major contributions: adaptive and progressive partitioning, exploitation of fragment correlations, and overlapping partitioning.

We generate instances of size 100GB and 500GB, both with uniform distribution, for the synthetic workloads. Table 3.1 shows parameters that we vary in the experiments as independent variables. The default value for each variable is shown in bold. We use the default value for the experiments unless otherwise mentioned. We consider three different query selectivities: Small (S) means that the selection condition returns 1% of the data; Medium (M) means that the selection condition returns 5%; and Big (B) means 25%. We use three different distributions for selection conditions of queries: uniform distributed (U), lightly skewed (L), and heavily skewed (H). Uniform means that for a fixed interval size, we pick a set of intervals such that the mid-point of the intervals is uniformly distributed. Lightly skewed means the mid-point of the selection intervals follows a normal distribution over the domain with a variance set to 7.5% of the domain. Heavily skewed also uses a normal distribution, but with the variance set to 0.25% of the domain.

Our evaluation is conducted on a cluster of 32 nodes. One node is a dedicated master node with 8 threads and 48GB memory. Each of the remaining 31 slave nodes has 6 threads, 12GB memory, and a 400GB disk. All results are based on the average of at least three runs, unless mentioned explicitly.
3.10.1 Workload for a Real-Life Application

We demonstrate two key properties of our system on a real-life application: 1) we compare the performance of DeepSea when there is no size limit for the materialization pool to Hive that does not uses materialization and NP, a materialization strategy that stores each view without partitioning them; 2) we compare the performance of DeepSea when there is a size limit for the pool to state-of-the-art view selection strategies such as the one of Nectar [47].

We create a histogram over the values of attribute ra for the table PhotoPrimary of SDSS. We then generate a BigBench dataset, and for all tables that contain attribute item_sk use the histogram that we obtained from SDSS attribute ra to sample values for item_sk. Furthermore, we generate a query workload: we pick ten query templates (Q1, Q5, Q7, Q9, Q12, Q16, Q20, Q26, Q29, Q30) from BigBench that contain joins, and we add a selection on attribute item_sk to these templates. We randomly pick 1000 selection ranges from the SDSS workload (selections on attribute ra of the table PhotoPrimary, kept in order of the query submission time). Next, we randomly picked a BigBench query template and mapped the selections of SDSS to selections on item_sk of the BigBench queries. Thus, we obtain a workload of 1000 BigBench queries simulating SDSS access patterns over an SDSS data distribution to evaluate the overall performance of DeepSea.

In this experiment, we compare DeepSea with two baselines. The first is the unmodified Hive system (H in the graphs). The second is a materialization strategy that does not use partitioning. We call this strategy non-partition (or NP in the graphs). This is akin to using a materialization strategy like ReStore [38]. However, in constrast to ReStore which only uses physical matching, NP applies our logical matching technique. Figure 3.3a shows performance results for the 500GB dataset without a pool size limit. Our approach requires only 64.2% of the time of non-partition materialization to execute the whole workload. Materialization without partitioning results in roughly 65.6% of the time of Vanilla Hive.

To evaluate the effectiveness of DeepSea’s selection strategy, we compare it with the view selection strategy of Nectar [47]. Nectar does not consider accumulated benefit as a factor. To understand the performance gain due to the use of accumulated benefit in contrast with the
Figure 3.3: Workload simulating SDSS (1000 queries), 500GB
other innovations in DeepSea, we extended Nectar’s cost-benefit model to include the accumulated benefit of a view or fragment. The modified cost-benefit measure $N^+$ for views which we call Nectar+ is: 

$$N^+(V) = \frac{\text{COST}(V) \times N(V)}{S(V) \times \Delta T}$$

where $\Delta T$ is the time elapsed since the last access to $V$ and

$$N(V) = \sum_{Q \text{ used } V \text{ at } t} (\text{COST}(Q) - \text{COST}(Q/V))$$

For fragments, we adapt our formula from Section 3.7.1 in a similar fashion by removing the application of the decay function. Figure 3.3b shows results for Nectar (N in the graph), Nectar+ (N+ in the graph), and DeepSea (DS in the graph) for different pool size limits. We observe that Nectar+ consistently outperforms Nectar, and DeepSea consistently outperforms Nectar+. When the size limit of the materialized view pool is relatively large (500GB, which is the total size of all base tables), the difference between Nectar, Nectar+ and DeepSea is marginal. When the size limit is shrunk to 10% of the total size of all base tables, DeepSea shows its strength requiring only $\sim 28\%$ of the time of Nectar (20% faster than Nectar+) and being 30% faster than Vanilla Hive. DeepSea keeps fragments that can improve the overall performance in the pool, because they are neighbors of more frequently accessed fragments. Nectar and Nectar+, however, evict these fragments because of their low hit count. When the pool limit is decreased to 5% of the total database size, all three techniques perform poorly (worse than original Hive with no materialization (Figure 3.3a)). This is because with such a small materialized view pool, all three strategies evict fragments that are accessed earlier and admit fragments that are accessed more recently. Since evicted fragments may be accessed soon after eviction, there is an “oscillation” in the pool with extra working being done for the materialization and little or no gain seen from this extra work.

3.10.2 Adaptive and Progressive Partitioning

To understand the benefits of partitioning strategy, we compare DeepSea with equi-depth partitioning (E in the graphs or E followed by a number indicating the number of fragments).
Equi-depth is a simple, non-adaptive and non-progressive alternative to DeepSea’s partitioning approach. To evaluate the benefit of progressive partitioning standalone, we tease out the benefits of using DeepSea which is workload aware.

For this experiment, we do not bound the size of the largest fragment. We use instances of query template Q30 and vary the selection condition of this query to produce workload sequences where $Q_{30,i}$ denotes the $i^{th}$ query in a sequence.

First we generate a sequence of queries that has small selectivity and is heavily skewed as defined at the beginning of this section. Figure 3.4 shows the cost of partitioned view creation and the cost for queries that reuse fragments. Figure 3.4a shows that when the number of generated fragments increases, the cost for creating and partitioning the view increases as well. In Figure 3.4b, we notice that if the same number of fragments are generated by both approaches (6 fragments in this experiment), equi-depth performs worse than DeepSea because of the larger size of fragments that must be read during query evaluation. Increasing the number of generated fragments for equi-depth reduces the average runtime for the following queries. However, when we set the number of fragments to be relatively large (60 fragments), performance decreases. Small fragment size affects performance negatively, because a large number of files has to be read and data is unevenly distributed among tasks. Figure 3.4c shows the cumulative time for the query sequence.

In addition to better performance, DeepSea also differs from equi-depth partitioning in terms of the execution of MapReduce jobs on the cluster. We analyze cluster utilization for the queries that reuse the generated fragments by running the default query sequence on the default dataset. Besides noticing the time needed in DeepSea is about 20% less than equi-depth, the number of map tasks issued to the Hadoop engine is about 40% to 50% more for equi-depth. The reason is that the fragments used by equi-depth to answer the query are larger than the ones used by DeepSea. Thus, the Hadoop engine issues more map tasks to parallelize the read as much as possible. This indicates that equi-depth uses more resources than DeepSea to answer the same queries.

We now investigate how characteristics of the workload affect the performance of DeepSea
Figure 3.4: Comparing equi-depth vs. adaptive partitioning (DeepSea) over workload using 10 instances of query template Q30, 100GB
compared to non-adaptive partitioning approaches such as equi-depth. In addition to measuring time for running a workload of 10 such queries, we also project the time (using linear regression) for 100 queries. Figure 3.5 shows the performance of materialization without partitioning (NP), materialization using an equi-depth partition of a fixed size (E), and our DeepSea approach using workload-aware partitioning (DS) compared to Hive on a 500GB dataset. The setting are indicated by concatenating the abbreviations for the selectivity and query-skew setting, for example, ML stands for medium selectivity and a lightly skewed distribution over the selection ranges.

Figure 3.5a shows that both partitioning techniques (DeepSea and equi-depth) perform well compared to Hive and non-partition materialization in all experiments. When the selectivity is large, (indicated by B), our partition techniques can save 50 to 60% compared to Hive. For medium (M) selectivity, the partition techniques can save 60 to 70% and for small (S) selectivity the partition techniques can save 70 to 80%. Materialization alone without partitioning (NP) provides only a 15 to 25% improvement over Hive.

For uniformly distributed selections, DeepSea, as expected, does not provide a performance improvement over an equi-depth strategy (E). This is because equi-depth is tailored for such a distribution and the adaptive techniques of DeepSea do not pay off. However, for lightly skewed and heavily skewed selections, DeepSea has a noticeable advantage (up to 30%) over equi-depth partitioning. The performance of DeepSea increases and that of equi-depth decreases when introducing more skew (switching from uniformly distributed to lightly skewed and heavily skewed workloads). This is because we use the same number of fragments for DeepSea and equi-depth. When the workload is more and more skewed, there are fewer and smaller fragments needed by DeepSea to get the same benefit achieved by equi-depth.

Most optimizers will push down selections for reducing the size of intermediate results. Our materialization strategy requires that selections are not pushed down and hence we incur a performance hit initially. But even for small selectivities, this cost is quickly amortized over a workload. To understand when the additional work DeepSea does (by not pushing selections) is worth the cost, we plot the number of queries needed to recoup the cost of DeepSea in
Figure 3.5b. Notice that for both DeepSea and equi-depth partitioning, the cost of not pushing a selection is recouped at almost the same point unless the workload is heavily skewed and includes queries with a large selectivity (requesting large portions of the data) in which case DeepSea has an advantage.

### 3.10.3 Exploitation of Fragment Correlations

We now compare our selection strategy that exploits fragment correlations against Nectar’s strategy that is oblivious of such correlations. We use a workload that consists of ten queries (template Q30) that have big selectivity and are heavily skewed followed by another ten queries (also template Q30) that have small selectivity and are heavily skewed. We use a 500GB dataset with the pool size limit set to 7GB. Figure 3.6 shows that DeepSea benefits from smoothing the distribution of hits to fragments from the same partition and, thus, is more likely to keep fragments that are similar to frequently accessed fragments.

Recall that we smoothen the distribution of hits over an attribute’s range by fitting it to a normal distribution. Figure 3.6 and Figure 3.7 show how the performance of our approach is affected by the distribution underlying the selections in a workload. DeepSea significantly outperforms Nectar’s selection strategy if the real hits follow a normal distribution. Importantly, it does not perform worse than Nectar if the selection ranges follow a radically different distribution (Zipf).

### 3.10.4 Overlapping Partitioning

A key benefit of overlapping partitioning is that it writes less data when repartitioning for certain patterns that we observe in real-life applications frequently. In order to compare overlapping partitioning with horizontal partitioning, we generate a workload sequence of 30 queries from template Q30 with small selectivity and heavy skew. The selections of Q30_1 to Q30_10 have a midpoint of 20,000, the selections of Q30_11 to Q30_20 have a midpoint of 40,000, and the selections of Q30_21 to Q30_30 have a midpoint of 60,000. The domain of the selection attribute is [0, 400,000]. We generate this workload to simulate the common query selection
Figure 3.5: Varying selectivity and skew, Q30, 500GB
Figure 3.6: Selection ranges following Normal distribution

Figure 3.7: Selection ranges following Zipf distribution
pattern that we have observed in SDSS.

We are switching the pattern between Q30_10 and Q30_11 and between Q30_20 and Q30_21. Figure 3.8 shows that overlapping partitioning is more robust against changes in the workload, because it avoids writing a fragment that extends from the current upper bound of the selections to the upper bound of the domain that has not been queried yet.

We also generated a workload with 200 queries using query template Q5, all of which have big selectivity and are heavily skewed. The selection ranges for the first 100 queries were sampled from one distribution while the selection ranges for the next 100 queries follow a different distribution. Running this workload on the 100GB dataset, we compare against materialization without partitioning (NP in the graph), equi-depth partitioning with 5 fragments (E-5 in the graph) and DeepSea with no repartitioning (NR in the graph). Figure 3.9a shows for changing workloads, DeepSea outperforms the non-progressive approach that never repartitions by 7% and equi-depth partitioning by 27%. Figure 3.9b shows the cumulative time of DeepSea normalized to the cumulative time of the NR approach (no repartitioning), from query 101 (the first query following the new distribution) to query 200. DeepSea performs worse than NR for the first 30 queries because of the cost of repartitioning. This cost, however, is amortized by
3.11 Conclusions

In this chapter, we studied an instance problem of Chapter 2 where workloads are read-only, and applied the solution in Chapter 2 to the instance problem. We introduced DeepSea, which is the first adaptive, progressive, workload-aware approach for automatic materialization and partitioning of views. Our cost-benefit model for both views and fragments takes the correlations among fragments into account. Our progressive partitioning accommodates both dynamic analytic workloads and exploratory workloads where users explore multiple regions in the data before finding (and then focusing on) a region of interest. DeepSea is implemented in Hive and our experiments demonstrate that our approach is more effective than traditional materialization techniques that do not consider the physical design of materialized views or do not adapt online to the workload. We also demonstrate that for real-life workloads, our view/fragment selection strategy outperforms state-of-the-art selection techniques when the materialized view pool has a small size limit.

In the next chapter, we will study another instance problem: workloads containing operations of both read and write. We will show how the general problem and its solution in Chapter 2 can be applied to the instance problem to improve the workload performance.
Figure 3.9: Adaptation to workload changes, Q5, 100GB
Chapter 4

An Online Data Placement Design for Polystore Systems

4.1 Introduction

In the previous chapter, we have shown an instance of the problem of combining physical design and materialization defined in Chapter 2: improving performance for non-update workloads. In this chapter, we will show another instance of this problem, i.e., improving performance for workloads containing updates. We will study how to apply the solution discussed in Section 2.6 and improve the performance of hybrid workloads that include data ingestion, update and analysis. We assume the workloads are executed in a heterogeneous distributed system such as Polystore [39], but we argue that our solution is applicable to standalone systems such as HTAP [18] as well.

In many modern applications, e.g., the Internet of Things (IoT), time-sensitive data generated by a large number of diverse sources must be collected, stored, and analyzed in a reliable and scalable manner. This is critical to supporting accurate and timely monitoring, decision making, and control. Traditional data warehousing architectures that have been based on separate subsystems for managing operational (OLTP), data ingestion (ETL), and analytical (OLAP) workloads in a loosely synchronized manner are no longer sufficient to meet these
needs. As a result, new approaches such as data stream warehousing [45], near real-time ware-
housing [83], lambda/kappa architectures [3, 4, 41], and HTAP systems [18] have recently
emerged. While these approaches architecturally differ from one another, low-latency data in-
gestion (a.k.a., streaming or near-real-time ETL) is seen as a critical component of the solution
in all of them.

In a recent work, we have designed and built a one-of-a-kind transactional stream pro-
cessing system called S-Store [66]. S-Store is a scalable main-memory system that supports
hybrid OLTP+streaming workloads with well-defined correctness guarantees including ACID,
ordered execution, and exactly-once processing [81]. While S-Store can be used as a stand-
alone system to support streaming applications with shared mutable state [24], it has been
shown that within the context of the BigDAWG Polystore system [39], S-Store can uniquely
enhance OLAP-style data warehousing systems with near real-time capabilities [67].

We believe that streaming ETL in particular stands out as the killer app for S-Store [65].
More specifically, S-Store can easily be programmed to continuously ingest configurable-size
batches of newly added or updated data from a multitude of sources, and apply the neces-
sary cleaning and transformation operations on them using its dataflow-based computational
model. Furthermore, it provides the necessary scalable system infrastructure for processing
ETL dataflows with transactional guarantees. A crucial component of this infrastructure is the
database-style local in-memory storage. S-Store’s storage facilities can be used for multiple
different purposes, including: (i) temporary staging of newly ingested batches and any inter-
mediate data derived from them as they are being prepared for loading into the back-end data
warehouse, (ii) caching copies of older data fragments from the warehouse that will need to
be frequently looked up during ETL, (iii) serving as the primary storage for data fragments
which are subject to frequent updates. In general, since our streaming ETL engine has all the
capabilities of an in-memory OLTP database, it can take over some of the responsibility of the
back-end warehouse. For example, it can be directly queried to provide fast and consistent
access to the freshest data.

Figure 4.1 shows a high-level overview of the streaming ETL architecture that we envi-
sion. All data newly collected from the sources (time-ordered, append-only streams as well as arbitrary insertions in general) and requests for in-place updates or deletions on older data are ingested through a transactional streaming engine (S-Store). The streaming engine in turn populates a back-end OLAP engine with updates on a frequent basis, through a data migration component. The migration component is bi-directional, i.e., data can be copied or moved between the two engines transactionally, in both directions. Meanwhile, all OLAP query requests to the system are received by a middleware layer that sits on top of the two engines. This layer maintains a global system catalog, which keeps track of all the data fragments and where they are currently stored. Based on this information, it determines where to forward the query requests for execution. Query results received from the engines are then returned back to the user. We have built a working prototype for this architecture based on Kafka [58] (data collection), S-Store (streaming ETL engine), Postgres (OLAP engine), and BigDAWG (migration + middleware layer) [65]. This architecture raises a number of interesting research issues in terms of cross-system optimization.

In this work, we study how different data placement strategies perform in the presence of
mixed (read and write) ETL workloads: Given a data fragment (i.e., the lowest level of data granularity - part of a relation), it can be stored in the streaming engine, in the OLAP engine, or in both. While the main-memory streaming engine can generally handle look-ups and updates faster, it has a limited memory budget. In contrast, the OLAP engine has larger storage capacity, but is slower to access. Furthermore, both engines are subject to dynamically changing workloads which consist of ingestion and query requests. Thus, given a mixed workload with different types of data, operations, and performance needs, data ingestion is affected greatly by the decisions of (i) which data fragments to store in the streaming engine, (ii) whether to copy or move the data fragments between the database engines, and (iii) whether to evict the data fragments from a database engine. As we will illustrate based on our experiments on the TPC-DI benchmark [74], this decision can have significant impact on ETL latency.

We would like to point out that the problem of data placement and the solution that we study in this chapter are not limited to distributed systems such as Polystore. Frameworks like HTAP have a similar design issue. For instance, data can be stored in different formats (row-store or column-store) in HTAP. For a mixed workload, we must design the formats for the data, and there may be a requirement to change the data format to improve the performance when the workload evolves. We can consider data stored in one format for HTAP as a database engine for the distributed system, e.g., data stored in row-store format can be considered as data stored in a streaming engine in Polystore, data stored in column-store format can be considered as data stored in an analytical engine in Polystore, and the data stored in one format can switch to another format, just like the data stored in database engine can be migrated to another. We argue that our techniques are applicable to any frameworks where there are multiple physical designs that can affect the performance of workloads containing both read and write operations. We will explain this in more detail in the following sections.

Our major contributions in this work are:

1. We proposed the first online data allocation approach for polystore ingestion.

2. We developed a cost model that considers a number of key factors that affect the performance depending on the types of operations in a workload.
3. We implemented our cost model and algorithms in an open source polystore system (Big-DAWG).

4. We experimented our techniques on both real-life and synthetic datasets and showed that our approach significantly improves the performance comparing to static data placement designs.

4.2 Related Work

Federated databases

Database management systems (DBMS) permit the decentralization of databases [51, 64, 77]. A federated DBMS consists of a set of logical components, and these components are tied together by a set of federated schemas. A federated controller is utilized for remote data access as a coordinator and translator. The physical design of a federated DBMS is dominated by the design of its components for best performance for local access. On the federated level, the design is to build federated schemas relating to component schemas, and how logical redundancy should be handled to achieve performance or reliability. Our work provides a solution for design on the federated level. Garlic [48, 56] proposes a query optimizer for federated databases, which is built on top of a set of heterogeneous databases and provides a unified interface for users so an issued query is parsed and a cost-based execution plan is generated considering different capabilities of the underlying databases. The problem solved in Garlic is orthogonal to our work. We study how different data placement strategies should be considered in the query optimizer based on the characteristics of the workload.

Distributed databases (distribution design)

Distribution design, one of the major challenges for distributed databases [70], studies how to fragment and place data in a distributed database system. There are two approaches (top-down and bottom-up) for different scenarios [23]. Bottom-up is usually applied to multi-database systems as an aggregation of existing databases. The main goal is to identify the data that are shared in existing databases and the difference of the data in these databases. Top-down is applied to systems from scratch and is more relevant to our approach. The top-down approach consists of two aspects: fragmentation
and allocation. Fragmentation studies how to partition data, while allocation investigates how to distribute the partitioned data (can be replicated) across the sites of the distributed database. These two aspects should be considered as an integrated process since the input of allocation is the output of fragmentation. Due to the complexity of the combined problem, a practical process is to measure the effectiveness of a given design (can be done by a human designer) and thus help the design to evolve for better performance. Our work is different in that we automate the process of allocation in the query optimizer in an online fashion.

Mukkamala et al. [68] describe a methodology for data allocation in distributed databases, which contains two phases: 1) partitioning data items into groups, and 2) assigning groups to nodes. For the first phase, they used a greedy algorithm to minimize data transfer between groups for all queries subject to group storage constraints. The data groups are then allocated to the nodes to satisfy the condition for reliability in phase 2. They do not consider database engines based on different data models, and do not work in an online fashion.

Blankinship et al. [19] propose to iteratively search the minimum cost for query optimization and the minimum cost for data allocation through greedy algorithms because of the cost dependency of these two techniques. Data replication is not considered in their approach.

Materialization for opportunistic design MISO [60] studies how to use materialization to speed up query execution in a federated system that contains an analytical engine for non-structured data (typically Hadoop [1]) and an analytical engine for structured data (e.g., MySQL). For structured data and queries that can be executed in RDBMS, often traditional RDBMS executes the query more efficiently than Hadoop. LeFevre et al. [60] investigate how to use the traditional RDBMS to execute some queries in the workload, and design the materialization plan for both the non-structured database engine and the RDBMS, considering the constraints such as the storage space and the bandwidth of transferring data between two engines. Different from their problem setting, we mainly consider data update for the streaming engines in a polystore system, and we also consider how the difference between moving data and caching data changes our design.

Auto-Tuning Database auto-tuning has abundant literature [20, 21, 22, 25, 26]. It has been
applied to index selection [13, 76], materialization [10, 11, 17] and partitioning [15, 37, 63, 71, 75]. Unlike the above applications, our online approach works for allocation of data on different database engines.

**Anti-Caching** Traditional caching stores a duplicate copy of data from disk in memory. Anti-Caching [33], designed for memory-based database systems, moves “cold” data from memory to disk (i.e., stored in memory format). There is only one copy of the data anytime for anti-caching, i.e., it is either stored in memory or on disk, and thus there is no need for synchronization. Anti-caching is designed for standalone databases; our work focuses on design for polystore systems.

### 4.3 Background

In this section we introduce the necessary background information for our system (BigDAWG and S-Store).

#### 4.3.1 BigDAWG Polystore

When it comes to database systems, it is commonly believed that “one-size no longer fits all” [79]. Specialized databases have become the norm. Some systems are designed specifically for unique types of data such as arrays or graphs [29, 34]. Others specialize in data formatting such that analytical queries can run extremely quickly [78]. Many workloads, however, require multiple of these specializations to execute efficiently.
Intel’s BigDAWG represents a polystore of multiple disparate database systems, each of which specializes in one type of data (e.g., relational, array, streaming, etc.) [39]. BigDAWG provides the user with querying and data migration across these systems, essentially abstracting the individual systems into one unified front-end from the user’s perspective. BigDAWG accomplishes this by separating databases into several “islands of information”, each of which contains multiple systems that share a common query language. For instance, relational databases such as Postgres and MySQL are connected in a “relational island”, which is queried via SQL statements (Figure 4.2).

While operations are delegated to the appropriate specialized system, BigDAWG also contains the ability to run queries on one engine using data from another engine. To facilitate this, BigDAWG contains the ability to efficiently migrate data from one engine to another. One specific scenario that data migration makes possible is the ingestion of streaming data into an analytical data warehouse. Such a scenario is best served by a data stream management system performing data cleaning operations on the streaming data before migrating the data to the OLAP engine, as discussed in Section 4.1.

### 4.3.2 S-Store

S-Store is a streaming system that specializes in the correct management of shared, mutable state [24, 66]. It is common for stream processing systems to define computations over streams as dataflow graphs. A dataflow graph is a directed acyclic graph (DAG). A node in the graph represents a streaming or a nested transaction, and an edge represents an execution ordering. S-Store models its dataflow graphs as a series of transactions, each of which has full ACID properties inherited from OLTP. S-Store also provides the ordering and exactly-once guarantees of a modern streaming system that ensures correctness [66].

S-Store is built on top of the main-memory OLTP system, H-Store [57]. Transactions are parameterized user-defined stored procedures, each of which passes output data along a stream in a dataflow graph to be used as the input parameters of the next downstream stored procedure. Each transaction atomically executes on a batch of tuples, the size of which is defined by the
user. Batches are ordered by their time of arrival, and that order is maintained throughout the dataflow graph. Transactions in a dataflow graph typically execute independently, meaning locks on shared state are released between consecutive transactions in the graph.

4.4 Motivation

In our data warehousing setting, the workload consists of a mix of ingest requests and query requests. These requests must be served on a continuous basis with low latency and high throughput. Ingest requests may generally consist of changes to the data in the warehouse including insertions, in-place updates, or deletions. Feeds from streaming data sources (e.g., sensors, stock market) are typically in the form of appends (i.e., only time-ordered insertions). Ingest requests are primarily served by S-Store, whereas query requests can be served by both S-Store or Postgres depending on the location of the needed data fragments.

S-Store transactionally processes ingest requests in small batches of tuples in order to apply ETL transformations. The resulting data fragments are then asynchronously migrated from S-Store to Postgres. This migration can be in the form of periodic pushes from S-Store to Postgres, or on-demand pulls by Postgres. Since S-Store has its own local storage, it can store a data fragment temporarily until migration, or even thereafter if that fragment will be needed by S-Store. While the natural direction of migration for newly ingested data is from S-Store to Postgres, we also support migration in the opposite direction. It can be useful to bring older data fragments back to S-Store, such as when an ETL transformation needs to look up older data for validating new data or when a certain data fragment starts receiving a burst of in-place updates on it.

In general, while a data fragment is being migrated from one engine (source) to another (destination), there are three options with respect to data placement:

1. **Move.** Delete the migrated data fragment from the source engine as part of the migration transaction (i.e., the destination engine becomes the one and only location for the fragment).
2. **Copy.** Continue to keep a copy of the migrated data fragment at the source engine (i.e., the fragment gets replicated at the destination engine).

3. **Evict.** Delete a data fragment from a database engine. This option has a prerequisite: the data fragment must exist in another database engine.

These options can have advantages or disadvantages under different circumstances. The **Move** option keeps a single copy of a data fragment in the system, which avoids redundant storage and, more importantly, removes the need to maintain transactional consistency across multiple copies in case of updates to the fragment. However, data access may take more time if the desired fragment is not available in local storage. On the other hand, the **Copy** option incurs an overhead for storing and maintaining multiple replicas of a data fragment in the system. This is largely due to the transactional overhead of two-phase commit. However, efficient, local data access to the fragment is guaranteed at all times. For the **Evict** option, it may bring benefit for updates because maintaining the consistency of the data is simplified, but meanwhile it forfeits the local data access guarantee.

While these are generic tradeoffs between **Move**, **Copy** and **Evict** for any given pair of engines, there are additional considerations specific to our setting. More specifically, our front-end processor, S-Store, and back-end warehouse, Postgres, differ in their system characteristics. Being a main-memory system, S-Store can provide fast data access, but has limited storage capacity. Furthermore, it is optimized for short-lived, read and update transactions (e.g., no sophisticated techniques for large disk scans). Postgres is disk-based, and can support large-scale, read-intensive OLAP workloads better than S-Store.

Although we focus on data placement in Polystore in this chapter, we can easily map the options described above to an HTAP system. The streaming engine in Polystore can be mapped to data stored in row-store format in HTAP because row-store format has better performance for data ingestion, and the analytical engine in Polystore can be mapped to data stored in column-store format in HTAP because column-store format has better performance for data analysis. **Move** in Polystore can be mapped to changing format in HTAP; **Copy** in Polystore
can be mapped to copying data to a different format in HTAP; and Evict in Polystore can be mapped to deleting data of a format in HTAP. By using the mappings we described, one can easily reduce the data placement problem for Polystore ingestion that we investigate in this chapter to a data configuration problem for HTAP (how to configure data format for a HTAP system to improve the performance of a hybrid workload).

In order to achieve high performance for a given mix of ingest and query workload, data fragments must be placed carefully. Both workload characteristics (e.g., frequency of reads vs. updates) as well as the tradeoffs discussed above must be taken into account. Furthermore, as the workload dynamically changes, placement of data fragments should be adjusted accordingly. Next, we will illustrate the problem using an example scenario taken from the TPC-DI benchmark [74].

Example 4.1. Streaming TPC-DI

TPC-DI is a data integration benchmark for evaluating the performance of traditional ETL tools [74]. It models a retail brokerage firm that needs to extract and transform data from heterogeneous data sources. The original benchmark does not involve streaming data. However, some of the data sources are incremental in nature and can be modeled as such. For example, Figure 4.3 shows the trade data ingestion portion of the benchmark remodeled as a streaming ETL scenario. In this dataflow, new trade tuples go through a series of validation and transformation procedures before they can be loaded into the DimTrade table of the warehouse.

A Case for Copy. One of those procedures (SP4) involves establishing foreign key dependencies with the DimAccount table. More specifically, when new rows are defined within the DimTrade table, reference must be made to the DimAccount table to assign the SK_AccountID key along with a few other fields. In other words, for each new batch of trade tuples to be ingested, the ETL dataflow must perform a lookup operation in the DimAccount table. Assume that an initial load for the DimAccount table has already been performed via S-Store (the ETL engine) before the Trade dataflow starts executing. In other words, DimAccount already resides in Postgres (the OLAP engine). Unless a copy of the DimAccount fragments were kept in S-Store after this initial load, SP4’s lookups would require migrating the relevant DimAccount
fragments from Postgres to S-Store. This in turn would incur high latency for the trade ingestion dataflow. In this scenario, keeping a local copy of DimAccount fragments to be referenced by the trade dataflow in S-Store would be a good data placement decision.

**A Case for Move.** Next, assume that S-Store occasionally ingests update requests for the DimAccount table. For DimAccount fragments that are replicated in S-Store, such updates must be transactionally applied on both engines in order to ensure mutual consistency. In this scenario, *Move* might be a more desirable data placement strategy for frequently updated DimAccount fragments than the *Copy* option. This way, S-Store would only have to locally update a single copy for those fragments.

**A Case for Evict.** Let us assume that DimAccount fragments are replicated in both S-Store and Postgres after a *Copy* is conducted. Similar to the case for *Move*, it may be beneficial to delete the fragments from Postgres, because update requests can be served more quickly by S-Store.

**OLAP Queries.** Now further assume that, while the above ingestion scenarios on DimTrade and DimAccount are taking place, a query request on the DimTrade table arrives. The query requires the system to scan the whole DimTrade table and calculate the total and average difference between bid and trade price values for each trade. Trade data is streaming into the system at high frequency and is being ingested into Postgres through S-Store via periodic, push-based migration. As such, the larger portion of DimTrade (which accounts for older
trades) is stored in Postgres, while the smaller, recently ingested portion is stored in S-Store. Therefore, this query cannot be answered in its entirety on either Postgres or S-Store. The query planner has multiple different options to ensure the most complete (and thus, the freshest) answer to this query. For example, Postgres can issue a pull request to migrate DimTrade fragments from S-Store and then execute the OLAP query in Postgres, or the middleware layer can execute the OLAP query on S-Store and Postgres in parallel and merge their results into the full answer. Some of these options have been analyzed in recent benchmark studies [65, 67]. The main takeaway is that data placement can also have significant impact on query performance. Therefore, the ETL workload must be considered in conjunction with the query workload in making data placement decisions.

4.5 The Data Placement Problem

In this section, we give formal definitions to the data placement problem.

4.5.1 Preliminaries

Before formalizing our problem statement, we give necessary definitions in this section.

At any point of time, the system manages a set of fragments of the tables. Assume a set \( T \) containing \( M \) tables \( T_1, \ldots, T_M \). Each table can be partitioned by a partition function \( Part_1, \ldots, Part_M \) into a number of fragments (e.g., horizontal partitioning as introduced in Section 3.3). We denote the fragments in \( T \) by \( F \), and use \( |F| \) to represent the number of the fragments. The set \( E \) contains two database engines \( S \) and \( A \), where \( S \) is a streaming engine (e.g., S-Store) and \( A \) is an OLAP engine (e.g., Postgres).

First we extend the definition for configuration (Definition 2.3) to define data placement in order to study the data placement problem. Definition 2.3 can be mapped to the configuration for the streaming engine because they both have a constraint for a storage size limit. The definition for data placement is broader than Definition 2.3 because the former contains the information of not only what data is stored in the streaming engine, but also what data is stored
in the analytical engine.

**Definition 4.1** (Data Placement). A data placement \( P \) is a matrix \( P_{|F| \times 2} \). Each element \( p_{ij} = \{0, 1\}, \) for \( 1 \leq i \leq |F|, 1 \leq j \leq 2 \).

In the above definition, each row of \( P \) represents the placement of a fragment. The first column represents the existence of a fragment in the streaming engine \( S \), and the second column represents the existence of a fragment in the OLAP engine \( A \). A zero represents that the fragment does not exist in the database engine, and a one represents that the fragment exists in the database engine. For instance, \( p_{31} = 0 \) denotes that the fragment \( f_3 \) does not exist in \( S \) and \( p_{32} = 1 \) denotes that \( f_3 \) exists in \( A \).

A data placement contains the information of what data fragments exist in what database engines before the execution of an operation. We use \( P_i \) to denote the data placement before the execution of \( Q_i \). There are several constraints for a data placement:

1. Similar to a constraint for configuration (Definition 2.3), there is an upperbound \( S_{\text{max}} \) for the storage size of the streaming engine in a data placement. The total size of the data fragments stored in the streaming engine in any data placement cannot exceed \( S_{\text{max}} \), or more formally, let \( \text{Size}(f_j) \) denote the size of fragment \( f_j \), \( \forall P_i \in P, \sum_{p_{j1}=1} \text{Size}(f_j) \leq S_{\text{max}} \).

2. \( \forall P_i, f_j \in F, \sum_{k=1}^{2} p_{jk} = \{1, 2\} \). This means a fragment must exist in at least one of the database engines.

Intuitively, we compute a sequence of data placements \( P_2, \ldots, P_n \) such that the workload performance is optimized subject to the above constraints. In order to effectively discuss this process, we illustrate the change of the data placement by a finite state machine shown in Figure 4.4.

In this finite state machine, there are three states and each state represents a possible placement of \( f_j \). For instance, \( \text{State}_S \) means \( f_j \) is in the streaming engine, and it is not in the analytical engine (i.e., \( p_{j1} = 1 \land p_{j2} = 0 \)); \( \text{State}_{S+A} \) means \( f_j \) is in both the streaming engine and the analytical engine (i.e., \( p_{j1} = 1 \land p_{j2} = 1 \)); and \( \text{State}_A \) means \( f_j \) is in the analytical
engine, and it is not in the streaming engine (i.e., $p_{j1} = 0 \land p_{j2} = 1$). As illustrated in Example 4.1, a state change from $States$ to $State_{A}$ is a **Move**, where $f_{j}$ is moved from the streaming engine to the analytical engine; a state change from $States$ to $States_{S+A}$ is a **Copy**, where $f_{j}$ is copied to the analytical engine from the streaming engine; a state change from $States_{S+A}$ to $States$ is an **Evict**, meaning that $f_{j}$ is evicted (removed) from the analytical engine. If the placement of $f_{j}$ is not changed, then the state points to itself.

We use $P_{i}$ to denote the data placement before the execution of $Q_{i}$. Now we extend the execution cost (Definition 2.4) and the configuration cost (Definition 2.5) for the data placement problem.

**Definition 4.2** (Execution Cost). We use $COST_{EXE}(Q, P)$ to denote the cost of executing operation $Q$ in data placement $P$.

**Definition 4.3** (Data Placement Cost). We use $COST(P_{i}, P_{i+1})$ to denote the data placement cost for $P_{i+1}$, i.e., the cost of migrating data between database engines from placement $P_{i}$ to $P_{i+1}$.

Similar to Definition 2.5, we assume the cost of removing a data fragment from a database engine is negligible, and thus when we compute the cost of a workload, we neglect this removing cost.
**Definition 4.4** (Operation Type). We consider three different types of operations in this work: Insert (ETL-read), Write (including in-place update and delete), and Read (OLAP-read). We denote this by OperationType(Q) = \{Insert, Write, Read\).

Insert is an operation that inserts a tuple into the system. As we have introduced, in our system, insert can only be executed in the streaming engine. Although the insert operation itself only inserts data into tables in the streaming engine and we do not discuss the operation in the context of data placement (details about migrating inserted data from the streaming engine to the analytical engine can be found in our recent work [65]), this operation often includes lookup operations (ETL-reads) in other tables in order to retrieve values in these tables through integrity constraints such as foreign keys. We illustrate this process by the following example.

**Example 4.2.** Continue the running Example 4.1, Table 4.1 describes the current status of the system. For simplicity, the table only contains a subset of the attributes of the tables in TPC-DI. On the left side, there are two fragments in the streaming engine: a DimTrade fragment \(f_2\) and a DimAccount fragment \(f_1\). On the right side, the analytical engine contains a fragment of the DimAccount table (\(f_0\)). Assume we are inserting a tuple \(t_1\) to the DimTrade table in the streaming engine. As illustrated in Example 4.1, a procedure (SP4) will try to find the foreign key dependencies with the DimAccount table. Assume the tuple to be inserted has TradeID=1 and AccountID=5. In order to insert this tuple into \(f_2\), we must retrieve the SK AccountID (10005) from the DimAccount fragment \(f_1\) by using the known AccountID (5). This lookup process is what we call an ETL-read, and it is a suboperation of an insert to the DimTrade table.

**Definition 4.5** (Executing Engines). Given an operation \(Q\) and a data placement \(P\), we denote the executing engines, the database engines in which \(Q\) is executed based on \(P\), by \(DB(Q, P)\).

Note that there can be more than one database engine involved in the computation of \(Q\). For instance, a write operation \(Q\) is executed in both \(S\) and \(A\) because the fragment to which \(Q\) writes exists in both \(S\) and \(A\).
In this work, we make the following assumptions about what types of operations can be executed in which engine.

1. Insert can only be executed in $S$.

2. Write can be executed in either $S$, $A$, or both.

3. Read can be executed in either $S$ or $A$.

The first requirement is based on the assumption that the streaming engine is usually more efficient than the analytical engine for insert operations. Thus, as shown in Figure 4.1, we want to direct all insertions to the streaming engine. Depending on the placement of a fragment and the operation, Write can be executed on either the streaming engine or the analytical engine, or both; Read can be executed on either the streaming engine or the analytical engine (in this work, we do not consider the case that an OLAP-read is executed on both engines in parallel).

**Definition 4.6 (Required Fragments).** Given an operation $Q$, we use $F(Q)$ to denote the required fragments, which is the set of the data fragments that are required by $Q$.

We now extend the definition for workload (Definition 2.1) to include insert operations for this chapter.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Tables in the system</td>
</tr>
<tr>
<td>F</td>
<td>Fragments of T</td>
</tr>
<tr>
<td>P</td>
<td>Data placement</td>
</tr>
<tr>
<td>E</td>
<td>The executing engines</td>
</tr>
<tr>
<td>S</td>
<td>The streaming engine (e.g. S-Store)</td>
</tr>
<tr>
<td>A</td>
<td>The analytical engine (e.g. Postgres)</td>
</tr>
<tr>
<td>S</td>
<td>The storage size limit of the streaming engine</td>
</tr>
<tr>
<td>W</td>
<td>The workload</td>
</tr>
<tr>
<td>H</td>
<td>The workload history</td>
</tr>
<tr>
<td>DB(Q)</td>
<td>The executing engines of operation Q</td>
</tr>
<tr>
<td>OperationType(Q)</td>
<td>The type of operation Q</td>
</tr>
<tr>
<td>F(Q)</td>
<td>The required fragments of operation Q</td>
</tr>
<tr>
<td>P</td>
<td>The data placement</td>
</tr>
</tbody>
</table>

Table 4.2: Variables and definitions

**Definition 4.7 (Workload)**. We use \( W \) to denote a workload, where \( \forall Q_i \in W, \text{OperationType}(Q_i) = \{ \text{Insert, Write, Read} \} \).

Table 4.2 lists the notation that will be used in the rest of this chapter.

### 4.5.2 Problem Statement

We now extend the problem definition in Section 2.5 to the the data placement problem of Polystore. For this problem, as we have discussed above, we extend the definition of configuration (Definition 2.3) to data placement (Definition 4.1), and accordingly we extend the configuration cost (Definition 2.5) to the data placement cost (Definition 4.3). For the purpose of simplicity, we use \( \text{COST}(Q, P) \) to denote the cost of executing operation \( Q \) in data placement \( P \). \( \text{COST}(Q, P) \) includes both the execution cost (Definition 4.2) and the data placement cost (Definition 4.3).

**Definition 4.8 (Online Data Placement)**. Assume we are given the input of \( W \) and the initial data placement \( P_1 \) defined as a constraint for Definition 4.1, we address the problem of Online Data Placement: find a sequence of data placement \( P = P_2, ..., P_n \) to minimize the total execution cost of the operations in \( W \)

\[
\text{COST}(W, P) = \sum_{i=1}^{n} \text{COST}_{\text{EXE}}(Q_i, P_i) + \sum_{i=1}^{n-1} \text{COST}(P_i, P_{i+1}) \quad (4.1)
\]
In Equation 4.1, $\text{COST}_{\text{EXE}}(Q, P)$ denotes the cost of executing operation $Q$ given placement $P$, and $\text{COST}(P_i, P_{i+1})$ denotes the cost of generating placement $P_{i+1}$ from placement $P_i$.

We note that for any operation $Q_i$ executed in a database engine ($S$ or $A$), if $Q_i$ is insert (ETL-read) or read (OLAP-read) and $f_j \in F(Q_i)$, but $f_j$ is not in the database engine, then $\text{COST}(Q_i, P_i)$ must include the cost of migrating $f_j$ to the database engine.

### 4.6 Solutions

In this section, first we give the overview of our solution to Online Data Placement by applying and extending the system process described in Section 2.6, then we introduce the assumptions and heuristics that we use and discuss the cost model based on these assumptions to solve the problem. Last we introduce the online algorithm based on the cost model.

#### 4.6.1 Solution overview

Data placement in federated systems is a NP-hard problem [70]. In our setting, Online Data Placement has at least the same complexity and thus is also intractable. In order to make the solution scalable, we design an approximated approach by applying and extending Section 2.6. Our approximated approach is based on a greedy algorithm.

As we have introduced, data placement (Definition 4.1) is an extension of configuration (Definition 2.3). Note for Online Data Placement, some sub-problems described in Section 2.6 can be solved trivially. For instance, using configurations can be solved by checking if a required data fragment exists in the executing engine, while determining candidate configurations can be solved by enumerating all possible different data placements as being shown in the explanation of Definition 4.1. We skip the procedures for these sub-problems. The major sub-problem we solve in this chapter is selecting configurations (determining the next data placement).

Algorithm 4.1 shows how an operation $Q$ in $W$ is executed. As we described above, we first
Algorithm 4.1 ExecuteOperation \((\mathcal{H}, Q, P, \text{STAT})\)

Input: Workload history \(\mathcal{H}\), Operation \(Q\), Data Placement \(P\), Data Statistics \(\text{STAT}\)

\begin{algorithmic}[1]
\State \(F_W \leftarrow \text{Fragments}(\mathcal{H})\)
\State \(F_Q \leftarrow \text{Fragments}(Q)\)
\For {each \(f_j \in F_Q\)}
\State \(\text{UPDATESTATS}(f_j, \text{OperationType}(Q), \text{STAT})\)
\EndFor
\EndFor
\If {\(\text{OperationType}(Q) \in \{\text{Insert}, \text{Read}\}\)}
\For {each \(f_j \in F_Q\)}
\If {\(\text{MIGRATIONREQUERED}(f_j, P) == \text{True}\)}
\State \(\text{MIGRATEFRAGMENT}(f_j, \text{DB}(Q), P, \text{STAT})\)
\EndIf
\EndFor
\ElseIf {\(\text{REDESIGN}(\mathcal{H}, Q) = \text{True}\)}
\State \(\text{DATAPLACEMENT}(F_W, P, \text{STAT})\)
\EndIf
\EndIf
\State \(\text{EXECUTE}(Q)\)
\State \(\text{UPDATECACHE}(P)\)
\end{algorithmic}

solve the subproblems of using configurations and determining candidate configurations (sub-procedure matchDesign, selectRewriting, and determineCandidates in Algorithm 2.1). These subprocedures are trivial in this chapter and we skip them in Algorithm 4.1 for simplicity.

Algorithm 4.1 describes how subprocedures selectDesign and instrumentOperation in Algorithm 2.1 are implemented. We update the statistics of the fragments that are accessed in \(Q\) (lines 3-4, details will be introduced in Section 4.6.2). Lines 3-10 describe our solution to subprocedure selectDesign in Algorithm 2.1. If the operation type is insert (ETL-read) or read (OLAP-read), we must check if all required fragments exist in the database engine executing \(Q\), and we start the fragment migration procedure to decide whether we should \textbf{Copy} or \textbf{Move} the missing fragments (lines 5-8). If \(Q\) is not an insert or read (\(Q\) is a write), \(Q\) can be executed in the fragments’ current engine without incurring a data migration. Unlike inserts and reads, we do not have to redesign the data placement for each write since the redesign process is expensive compared to the cost of a write. We can redesign the placement after a certain number of operations; we can also redesign the placement after a certain amount of time (lines 9-10). The choice among these options is beyond this work’s scope and we do not discuss it here. After we finish the data placement design, we execute \(Q\) and update the S-Store cache if necessary because we implement the storage budget of S-Store cache as a soft constraint (Section 4.5.2).

Having introduced the overview of our solution, we are ready to introduce the assumptions and heuristics in our setting and discuss the cost model based on these assumptions and the online algorithm for the problem of \textit{Online Data Placement}.
We make the following assumptions and heuristics to speed up the computation for data placement:

1. The operations in workload $\mathcal{W}$ is executed in order. We do not consider executing the operations in parallel when computing the cost of an operation for a data placement.

2. The cost of deleting fragment $f_j$ from a database engine is 0.

3. The cost difference between a write in Postgres and a write in S-Store is the cost of the write in Postgres. In our experiments we have observed that often the cost of executing a write in Postgres is far more expensive (more than one order of magnitude) than that in S-Store. Thus when we compute the cost difference between the same write operation in S-Store and Postgres, we simply take the write cost in Postgres.

4. The fragment accesses in workload $\mathcal{W}$ are independent from each other. This is often true for inserts (ETL-reads) and writes since these operations typically only access one fragment a time. However, depending on $Q$ and how data is partitioned, this assumption is sometimes unrealistic for reads (OLAP-reads), since the fragments that are involved in an OLAP-read are usually correlated (as we have shown in Chapter 3). Considering the correlation of the fragments can add more complexity to a problem that is already hard. We leave this to future work.

5. When we compute the cost and benefit of a data placement $P$ for a workload history $\mathcal{H}$, we assume $P$ remains unchanged while executing the operations in $\mathcal{H}$.

6. A fragment is accessed by an operation only once. This again is often true for inserts and writes, but it can be false for reads (OLAP-reads). The cost model will be more complicated without this assumption. We leave this to future work.

In the next section we will introduce our cost model that is based on the above assumptions.
### Variable Description

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_I(j) )</td>
<td>Number of insert (ETL-read) for ( f_j ) in ( H )</td>
</tr>
<tr>
<td>( N_W(j) )</td>
<td>Number of write for ( f_j ) in ( H )</td>
</tr>
<tr>
<td>( N_R(E, j) )</td>
<td>Number of read in database engine ( E ) for ( f_j ) in ( H, E \in {S, A} )</td>
</tr>
<tr>
<td>( C_R(E, j) )</td>
<td>Cost of read of ( f_j ) in database engine ( E, E \in {S, A} )</td>
</tr>
<tr>
<td>( C_W(E, j) )</td>
<td>Cost of write of ( f_j ) in database engine ( E, E \in {S, A} )</td>
</tr>
<tr>
<td>( C_M(SA, j) )</td>
<td>Cost of migration for ( f_j ) from ( S ) to ( A )</td>
</tr>
<tr>
<td>( C_M(AS, j) )</td>
<td>Cost of migration for ( f_j ) from ( A ) to ( S )</td>
</tr>
</tbody>
</table>

Table 4.3: Statistics in the cost model

#### 4.6.2 Cost model

Table 4.3 defines a fragment \( f_j \)'s statistics that are used in our cost model to optimize the execution of workload \( W \). We compute the cost (Equation 4.1) of executing \( Q_i \) in data placement \( P_i \) as follows:

1. if \( OperationType(Q_i) = \text{Insert} \),

\[
COST(Q_i, P_i) = \sum_{f_j \in F(Q_i), p_{j1}=1} C_R(S, j) + \sum_{f_i \in F(Q_i), p_{i1}=0} (C_R(S, l) + C_M(AS, l))
\]  

(4.2)

2. if \( OperationType(Q_i) = \text{Write} \),

\[
COST(Q_i, P_i) = \sum_{f_j \in F(Q_i), p_{j1}=1} C_W(S, j) + \sum_{f_i \in F(Q_i), p_{i1}=1} C_R(A, l)
\]

(4.3)

3. if \( OperationType(Q_i) = \text{Read} \land DB(Q_i, P) = \{S\} \),

\[
COST(Q_i, P_i) = \sum_{f_j \in F(Q_i), p_{j1}=1} C_R(S, j) + \sum_{f_i \in F(Q_i), p_{i1}=0} (C_R(S, l) + C_M(AS, l))
\]

(4.4)

4. if \( OperationType(Q_i) = \text{Read} \land DB(Q_i, P) = \{A\} \),

\[
COST(Q_i, P_i) = \sum_{f_j \in F(Q_i), p_{j2}=1} C_R(A, j) + \sum_{f_i \in F(Q_i), p_{i2}=0} (C_R(A, l) + C_M(SA, l))
\]

(4.5)

Equation 4.2 denotes the cost of an insert (ETL-read) operation \( Q_i \). The first term denotes the total cost of reading the required fragments that exist in the streaming engine, the second term denotes the total cost of reading the required fragments that do not exist in the streaming
engine (thus the cost of migrating the fragments from Postgres to S-Store must be included). Equation 4.3 to 4.5 are defined similarly.

We now define the execution cost of the operations that require \( f_j \) in the workload history \( \mathcal{H} \) if \( f_j \) is in one of the three states (recall that we assume the placement stays unchanged in \( \mathcal{H} \) when computing the cost).

1. \( State_S : p_{j1} == 1 \land p_{j2} == 0. \)

\[
C(S, j) = N_I(j) \times C_R(S, j) + N_W(j) \times C_W(S, j) + N_R(S, j) \times C_R(S, j) + N_R(A, j) \times (C_M(SA, j) + C_R(A, j))
\]  
(4.6)

2. \( State_A : p_{j1} == 0 \land p_{j2} == 1. \)

\[
C(A, j) = N_I(j) \times (C_M(AS, j) + C_R(S, j)) + N_W(j) \times C_W(A, j) + N_R(S, j) \times (C_M(AS, j) + C_R(S, j)) + N_R(A, j) \times C_R(A, j)
\]  
(4.7)

3. \( State_{S+A} : p_{j1} == 1 \land p_{j2} == 1. \)

\[
C(S + A, j) = N_I(j) \times C_R(S, j) + N_W(j) \times (C_W(S, j) + C_W(A, j)) + N_R(S, j) \times C_R(S, j) + N_R(A, j) \times C_R(A, j)
\]  
(4.8)

In the definition of \( State_S \) (Equation 4.6), the first term \( N_I(j) \times C_R(S, j) \) represents the cost of executing insert (ETL-read) for \( f_j \) in \( \mathcal{H} \); the second term \( N_W(j) \times C_W(S, j) \) represents the cost of executing write (in-place update and delete) for \( f_j \) in \( \mathcal{H} \); the third term \( N_R(S, j) \times C_R(S, j) \) represents the cost of executing read (OLAP-read) in \( S \) for \( f_j \) in \( \mathcal{H} \); the last term \( N_R(A, j) \times (C_M(SA, j) + C_R(A, j)) \) represents the cost of executing read (OLAP-read) in \( A \) for \( f_j \) in \( \mathcal{H} \) (Since \( p_{j2} == 0 \), for each such OLAP-read, \( f_j \) must be migrated to \( A \) first). \( State_A \) (Equation 4.7) and \( State_{S+A} \) (Equation 4.8) are defined similarly.

Choosing a data placement for \( f_j \) requires comparing the cost of Equation 4.6, 4.7, and 4.8. Example 4.3 shows how our cost model works for data placement design.

**Example 4.3.** Assume there are three fragments \( f_1, f_2, \) and \( f_3 \) in the system, and Table 4.4
describes the cost for each placement. The second column represents the current placement of the fragments ($f_1$ is in S-Store, $f_2$ is in both S-Store and Postgres, and $f_3$ is in Postgres), and we are now required to redesign the placement for each fragment. We examine the cost of each of the three possible placements for the workload history $H$, and choose the placement that brings the most benefit (the least cost for $H$, including the cost of necessary migrations that are required for the operations in $H$). We then migrate data for each of such fragments accordingly.

Assume that $C(S, 1) < C(P, 1) < C(S + P, 1)$, $C(S, 2) < C(P, 2) < C(S + P, 2)$, and $C(S + P, 3) < C(P, 3) < C(S, 3)$. For $f_1$, the minimal cost is to keep it in S-Store, therefore in the new placement, $f_1$ is kept unchanged (in S-Store). For $f_2$, the minimal cost is obtained by keeping $f_2$ only in S-Store. $f_2$’s current placement is S-Store and Postgres, it must be evicted from Postgres. Since we assume evictions incur no additional cost, we evict $f_2$ from Postgres. For $f_3$, the minimal cost is obtained by keeping the fragment in both S-Store and Postgres. The current placement of $f_3$ is Postgres. In order to change the placement for the minimal cost, we must check if $C(S + P, 3) + C_M(AS, 3) < C(P, 3)$. If it is, then $f_3$ is copied from Postgres to S-Store, otherwise we keep the state of $f_3$ unchanged since $C(P, 3) < C(S, 3)$.

We notice that in the above equations, there are many common factors. In the following discussion, instead of computing the total cost by using the above equations directly, we can simplify the process by computing the difference between these costs as the benefit of adopting one data placement against another, and the common factors are cancelled out during this process. For instance, to compute the benefit of having $f_j$ in S-Store against having it in Postgres, we simply subtract Equation 4.7 by Equation 4.6. If the result is greater than 0, it means the cost of keeping $f_j$ in S-Store is less than keeping it in Postgres. Next we will discuss updating data placement for $f_j$ by computing the benefit of placement changing, assuming $f_j$ is currently in either S-Store, Postgres, or S-Store and Postgres respectively.

Table 4.4: Cost changes between different placement

<table>
<thead>
<tr>
<th>Fragments</th>
<th>Curr. Placement</th>
<th>Execution Cost of $H$ for Next Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>$p_{11} == 1 ∧ p_{12} == 0$</td>
<td>$C(S, 1)$ $C(P, 1)$ $C(S + P, 1)$</td>
</tr>
<tr>
<td>$f_2$</td>
<td>$p_{11} == 1 ∧ p_{12} == 1$</td>
<td>$C(S, 2)$ $C(P, 2)$ $C(S + P, 2)$</td>
</tr>
<tr>
<td>$f_3$</td>
<td>$p_{11} == 0 ∧ p_{12} == 1$</td>
<td>$C(S, 3)$ $C(P, 3)$ $C(S + P, 3)$</td>
</tr>
</tbody>
</table>
Current placement: S-Store

First let us study the case that the current placement of \( f_j \) is \( p_{j1} = 1 \land p_{j2} = 0 \), i.e., \( f_j \) is in S-Store only. We would like to find out if we should Move or Copy \( f_j \) to Postgres, or keep \( f_j \) in S-Store unchanged. We define the benefits for Move and Copy \( f_j \) to Postgres as follows:

When we Move \( f_j \), the benefit of having \( f_j \) in Postgres is Equation 4.6 - Equation 4.7, i.e.,

\[
B(MV, SA, j) \approx -N_I(j) \times C_M(AS, j) - N_W(j) \times C_W(A, j) \\
-N_R(S, j) \times C_M(AS, j) + N_R(A, j) \times C_M(SA, j)
\]  

(4.9)

When we Copy \( f_j \), the benefit of having \( f_j \) in Postgres is Equation 4.6 - Equation 4.8:

\[
B(CP, SA, j) \approx -N_W(j) \times C_W(A, j) + N_R(A, j) \times C_M(SA, j)
\]  

(4.10)

Comparing Equation 4.10 and Equation 4.9, \( B(MV, SA, j) - B(CP, SA, j) = -N_I(j) \times C_M(AS, j) - N_R(S, j) \times C_M(AS, j) \leq 0 \), i.e. \( B(MV, SA, j) \leq B(CP, SA, j) \). This implies that we only have to check if \( B(CP, SA, j) \leq C_M(SA, j) \). If it is true, we keep the placement of \( f_j \) unchanged. If it is not, since \( B(MV, SA, j) \leq B(CP, SA, j) \), we always Copy \( f_j \) from S-Store to Postgres.

Current placement: Postgres

Now let us study the case that the current placement of \( f_j \) is \( p_{j1} = 0 \land p_{j2} = 1 \), i.e., \( f_j \) is in Postgres only. As we have discussed in Section 4.5, a specific constraint for S-Store is that there is a cache size limit (storage budget) for S-Store.

In order to satisfy the above constraint, we will use a greedy algorithm to decide what fragments to be kept in S-Store (introduced in Section 4.6.1). To define the benefit of keeping \( f_j \) in S-Store after a migration of \( f_j \) from Postgres to S-Store, we must consider two different scenarios: whether we Move \( f_j \) or Copy \( f_j \) from Postgres to S-Store. When we Move \( f_j \), the
benefit of having \( f_j \) in S-Store is Equation 4.7 - Equation 4.6, i.e.,

\[
B(MV, AS, j) \approx N_I(j) \times C_M(AS, j) + N_W(j) \times C_W(A, j) \\
+ N_R(S, j) \times C_M(AS, j) - N_R(A, j) \times C_M(SA, j)
\] (4.11)

When we **Copy** \( f_j \), the benefit of having \( f_j \) in S-Store is Equation 4.7 - Equation 4.8:

\[
B(CP, AS, j) \approx N_I(j) \times C_M(AS, j) + N_R(S, j) \times C_M(AS, j)
\] (4.12)

In the above equations, we assume \( C_W(A, j) - C_W(S, j) \approx C_W(A, j) \) as one of our heuristics introduced earlier in this section. We define the benefit of keeping \( f_j \) in S-Store as

\[
B(AS, j) = \max(B(MV, AS, j), B(CP, AS, j))
\] (4.13)

We will use Equation 4.13 to decide what fragments to be kept and what fragments to be evicted from S-Store (Section 4.6.3).

**Current placement: S-Store and Postgres**

Lastly we study the third scenario where the current placement of \( f_j \) is \( p_{j1} = 1 \land p_{j2} = 1 \), i.e., \( f_j \) is in both S-Store and Postgres. Since executing an update in S-Store is much faster than that in Postgres, we may want to evict \( f_j \) from Postgres if that brings benefit to \( H \). By comparing Equation 4.6 and Equation 4.8, we define the following Equation 4.14 to be the benefit of evicting \( f_j \) from Postgres.

\[
B(EV, A, j) = N_W(j) \times C_W(A, j) - N_R(A, j) \times C_M(SA, j)
\] (4.14)

When \( B(EV, A, j) > 0 \), we should consider evicting \( f_j \) from Postgres. We notice that there is no benefit to evict \( f_j \) from S-Store if \( f_j \) is in both S-Store and Postgres, because in our setting, for all of the three types of operations (insert, write, and read), keeping \( f_j \) in S-Store does not increase the cost, and thus we neglect this option.

Having introduced how we compute the cost and benefit for the changes of the different
states of the data placement, next we will discuss how we use decay functions to make the solutions adaptive to an evolving workload.

**Decay Function**

An online approach must be aware of the changes of the access patterns of a workload. As we have introduced in Chapter 3 and Chapter 5, one key component of our techniques is that we use decay functions to age out the benefits of the data accesses happened in the past for a data placement design, i.e., the earlier an access has happened, the less benefit of this access for a data placement design is counted. Our approach adjusts the data placement based on the changes of the workload. In this approach, after computing a new placement for a fragment for each operation, the statistics of the number of each type of operation that have accessed fragment $f_j$ ($N_I(j)$, $N_W(j)$, $N_R(S,j)$, and $N_R(A,j)$ that are introduced in Table 4.3) for all fragments must be refreshed according to the decay functions that are defined next.

Let us denote $N_I(j,t_{\text{now}})$ to the decayed number of $N_I(j)$, which is the number of inserts that read (ETL-read) fragment $f_j$ in $H$. We define $N_I(j,t_{\text{now}})$ as follows (the decayed numbers of the other three variables $N_W(j)$, $N_R(S,j)$, and $N_R(A,j)$ are defined in the same fashion):

\[ N_I(j,t_{\text{now}}) = \sum_{t \in (0, t_{\text{now}})} \text{DEC}(t_{\text{now}}, t) \text{ when } Q \text{ accessed } f_j \text{ at } t \]  

(4.15)

In Equation 4.15, $t_{\text{now}}$ is the current time, $t$ is the time that an operation $Q$ (an insert that includes an ETL-read of $f_j$) is executed. $\text{DEC}(t_{\text{now}}, t)$ is a monotonically decreasing function (in $t_{\text{now}} - t$) that maps the current time ($t_{\text{now}}$) and the time ($t$) when $Q_i$ was executed to a value in $[0, 1]$. $\text{DEC}(t_{\text{now}}, t)$ is used to weight past usage by their age, and it enables our approach to adapt to an evolving workload. In our implementation we use the decay function as follows:

\[ \text{DEC}(t_{\text{now}}, t) = \frac{t}{t_{\text{now}}} \]  

(4.16)

Although we could keep all the timestamps for each access of the operations, it would cost unnecessary storage space and extra computation. The definition of our decay function leads
the computation of \( N_I(j, t_{\text{now}}) \), \( N_W(j, t_{\text{now}}) \), \( N_R(S, j, t_{\text{now}}) \), and \( N_R(A, j, t_{\text{now}}) \) to a simplified form. For simplicity, we only discuss the case that the operation is an insert, the statistics for the other three cases can be derived trivially. Assume at time \( t_{\text{last}} \) there is the latest insert that accessed \( f_j \), and there are a decayed number of \( N_I(j, t_{\text{last}}) \) inserts having accessed \( f_j \) at \( t_{\text{last}} \), where \( t_{\text{last}} \leq t_{\text{now}} \). In order to compute the decayed number of inserts at \( t_{\text{now}} \), we can compute \( N_I(j, t_{\text{now}}) \) as follows:

\[
N_I(j, t_{\text{now}}) = \begin{cases} 
N_I(j, t_{\text{last}}) \times \frac{t_{\text{last}}}{t_{\text{now}}} & \text{if } f_j \text{ is not accessed by an insert at } t_{\text{now}} \\
N_I(j, t_{\text{last}}) \times \frac{t_{\text{last}}}{t_{\text{now}}} + 1 & \text{otherwise}
\end{cases}
\] (4.17)

Thus we keep only the timestamp and the decayed number of times that a fragment is accessed last, instead of keeping the timestamps for all accesses. We use Equation 4.17 to compute the decayed number of accesses when it needs to be re-evaluated at time \( t_{\text{now}} \).

**Proposition 4.1.** Equation 4.15 and Equation 4.17 are equivalent.

**Proof.** We prove Proposition 4.1 by using induction.

Basis: Assume \( f_j \) has not been accessed by any insert at time \( t_0 \), i.e.,

\[
N_I(j, t_0) = \sum_{t \in (0, \ldots, t_0) | \exists Q \text{ such that } Q \text{ accessed } f_j \text{ at } t} \text{DEC}(t_0, t) = 0
\]

It is obvious that \( N_I(j, t_0) = N_I(j, t_{\text{last}}) \times \frac{t_{\text{last}}}{t_0} = 0 \).

Induction: Assume \( f_j \) has been accessed \( n \) times by insert, the last insert accessed \( f_j \) at time \( t_{\text{last}} \), and Equation 4.15 and Equation 4.17 are equivalent at time \( t_1 \) (\( t_1 \geq t_{\text{last}} \)), i.e.,

\[
\sum_{t \in (0, \ldots, t_1) | \exists Q \text{ such that } Q \text{ accessed } f_j \text{ at } t} \text{DEC}(t_1, t) = \sum_{t \in (0, \ldots, t_1) | \exists Q \text{ such that } Q \text{ accessed } f_j \text{ at } t} \frac{t}{t_1} = N_I(j, t_{\text{last}}) \times \frac{t_{\text{last}}}{t_1}
\]

We want to prove that Equation 4.15 and Equation 4.17 are equivalent at time \( t_2 \) when \( t_2 \geq t_1 \). Assume there is no insert accessing \( f_j \) between \( t_1 \) and \( t_2 \) (including \( t_2 \), it can be
proved trivially if there is an insert accessing \( f_j \) at \( t_2 \). Equation 4.15 gives us:

\[
N_I(j, t_2) = \sum_{t \in (0, \ldots, t_2) \exists Qs.t. Q accessed f_j at t} \text{DEC}(t_2, t)
\]

Equation 4.17 gives us:

\[
N_I(j, t_2) = N_I(j, t_{last}) \times \frac{t_{last}}{t_2} = N_I(j, t_{last}) \times \frac{t_{last}}{t_1} \times \frac{t_1}{t_2}
\]

Since Equation 4.15 and Equation 4.17 are equivalent at time \( t_1 \), i.e.,

\[
\sum_{t \in (0, \ldots, t_1) \exists Qs.t. Q accessed f_j at t} \frac{t}{t_1} = N_I(j, t_{last}) \times \frac{t_{last}}{t_1}
\]

Since there is no insert accessing \( f_j \) between \( t_1 \) and \( t_2 \),

\[
N_I(j, t_2) = \sum_{t \in (0, \ldots, t_2) \exists Qs.t. Q accessed f_j at t} \frac{t}{t_1} \times \frac{t_1}{t_2} = \sum_{t \in (0, \ldots, t_2) \exists Qs.t. Q accessed f_j at t} \frac{t}{t_2}
\]

Therefore Equation 4.15 and Equation 4.17 are equivalent at time \( t_2 \). \( \square \)

Example 4.4 shows the process of using Equation 4.17 to calculate the decayed number of data accesses.

**Example 4.4.** Assume a workload starts at time 0, and fragment \( f_j \) is accessed by insert (ETL-read) operations at time 1, 2 and 6. We want to measure the decayed number of inserts (ETL-reads) of \( f_j \) at time 8, assuming there is no insert accesses \( f_j \) at time 8. By using the timestamps and Equation 4.15, we know the decayed number \( N_I(j, 8) = \frac{1}{8} + \frac{2}{8} + \frac{6}{8} = \frac{9}{8} \). Using Equation 4.17, we calculate the decayed number each time \( f_j \) is accessed by an insert. At time 1, the decayed number \( N_I(j, 1) = 1 \). At time 2, the decayed number \( N_I(j, 2) = N_I(j, 1) \times \frac{t_{last}}{t_{now}} + 1 = 1 \times \frac{1}{2} + 1 = \frac{3}{2} \). At time 6, the decayed number \( N_I(j, 6) = N_I(j, 2) \times \frac{t_{last}}{t_{now}} + 1 = \frac{3}{2} \times \frac{2}{8} + 1 = \frac{3}{2} \). At time 8, the decayed number \( N_I(j, 8) = N_I(j, 6) \times \frac{t_{last}}{t_{now}} = \frac{3}{2} \times \frac{9}{8} = \frac{9}{8} \). The result agrees with that is computed by using Equation 4.15. By using Equation 4.17, we do not have to record the timestamps (time 1, 2, and 6), instead we use the previous decayed number and the last time \( f_j \)
is accessed to calculate the refreshed decayed number.

### 4.6.3 Data placement procedures

In this section, we introduce the key procedures that are called in Algorithm 4.1. Table 4.5 lists the equations that will be used in the following procedures.

As we have discussed in Algorithm 4.1, we categorize the operations into two sets: one category is insert (ETL-read) and read (OLAP-read) that must be executed in the designated database engines, and the other category is write that can be executed in either database engine or both.

For the first category of operations, any missing fragments that are required by an operation must be migrated to the engine where the operation is executed. Assume $Q$ is designated to be executed in database engine $E$. Algorithm 4.2 describes how we decide between a **Move** or a **Copy** of fragment $f_j$ to $E$ when $f_j$ is required by $Q$ and $f_j$ is not in $E$. If the designated engine is S-Store, we estimate: (1) if the missing fragment $f_j$ could be kept in S-Store after the execution of $Q$; (2) if the benefit of a Move would be larger than that of a Copy. We Move $f_j$ to S-Store if the above two conditions are satisfied, and Copy $f_j$ to S-Store otherwise. If the designated engine is Postgres, we check if the benefit of a Copy $f_j$ is larger than that of a Move. If it is, we Copy $f_j$ to Postgres, otherwise we Move $f_j$ to Postgres.

For the second category of operations, i.e., write that can be executed in either database engine or both, we only redesign the data placement based on certain criteria that are based on either time or the number of operations. Although we do not discuss these criteria in this work, we argue that these different criteria can be trivially embedded into our system. Algorithm 4.3 shows the overview of how we conduct a new data placement for a set of fragments. For each of

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B(MV, AS, j)$</td>
<td>Equation 4.11</td>
</tr>
<tr>
<td>$B(CP, AS, j)$</td>
<td>Equation 4.12</td>
</tr>
<tr>
<td>$B(AS, j)$</td>
<td>Equation 4.13</td>
</tr>
<tr>
<td>$B(EV, A, j)$</td>
<td>Equation 4.14</td>
</tr>
<tr>
<td>$B(CP, SA, j)$</td>
<td>Equation 4.10</td>
</tr>
</tbody>
</table>

Table 4.5: Symbols and equations
Algorithm 4.2 Migrate Fragment Algorithm

procedure MIGRATE_FRAGMENT(f\textsubscript{j}, DB(Q), P, STAT)

if DB(Q) == S

if KEEP_INSTORE(f\textsubscript{j}, P, STAT) == True && \(B(MV, AS, j) > B(CP, AS, j)\) then

\(p_{j1} \leftarrow 1, p_{j2} \leftarrow 0\) \{Move to DB(Q)\}

else

\(p_{j1} \leftarrow 1, p_{j2} \leftarrow 1\) \{Copy to DB(Q)\}

else

if \(B(CP, SA, j) < 0\) then

\(p_{j1} \leftarrow 0, p_{j2} \leftarrow 1\) \{Move to DB(Q)\}

else

\(p_{j1} \leftarrow 1, p_{j2} \leftarrow 1\) \{Copy to DB(Q)\}

Algorithm 4.3 DataPlacement (F, P, STAT)

Input : Fragments F, Data Placement P, Data Statistics STAT
Output : Updated data placement P and statistics STAT

for each \(f\textsubscript{j} \in F\) do

UPDATE_PLACEMENT(f\textsubscript{j}, P, STAT)

EXECUTE_MIGRATIONS(f\textsubscript{j}, P)

the fragments, we check if updating the data placement is beneficial for executing the workload history \(H\) (introduced in Algorithm 4.4), and then execute necessary data migrations.

Algorithm 4.4 summarizes how we update the data placement of \(f\textsubscript{j}\). Procedure keepInS-Store checks if the benefit brought by migrating \(f\textsubscript{j}\) to S-Store is higher than the benefit of migrating other fragments.

4.7 System Prototype and Experimental Evaluation

We have created a prototype for streaming data ingestion [65]. This prototype uses a combination of BigDAWG and S-Store, in conjunction with Kafka (a publish-subscribe messaging system [58]) and the relational database Postgres (Figure 4.5). New tuples arrive from a variety of data sources and are queued in Kafka. These tuples are batched and pushed to S-Store. As a streaming system with ACID state management, S-Store is particularly well-suited for streaming data ingestion workloads. Streaming data can be ingested and transformed in a dataflow graph, with intermediate state being maintained in a transactional manner.

For each stored procedure that requires access to data, S-Store checks the data catalog in BigDAWG through a fragment selection module. Data catalog in BigDAWG maintains all information about data fragments including the data placement. If the required data fragment
only exists in Postgres, the fragment selection module will instruct the data migrator in BigDAWG to migrate the fragment from Postgres to S-Store. Meanwhile, the fragment selection module is also responsible for deciding whether the migration should be Move or Copy, which fragment(s) should be evicted if the total size of the fragments exceeds the storage limit of S-Store, and if we should Evict fragment(s) from Postgres to improve the performance for updates.

In order to guarantee transactional safety for data migrations, we implemented two-phase commit for the migrations between S-Store and Postgres by extending BigDAWG’s migration framework.

### 4.7.1 Evaluation Setup

In the implementation, data can be migrated between S-Store (the streaming ETL engine) and Postgres (the OLAP engine), and queries can be run on either system. We executed the experiments on an Intel® Xeon® machine with 64 virtual cores and 132GB memory. S-Store is co-located on the same node as Postgres for ease of communication and migration. We warmed up the S-Store cache for ten seconds before collecting statistics.

To motivate the use of multiple systems in tandem, we implemented a subset of TPC-DI as a streaming workload (Example 4.1). We consider two tables in this workload: DimTrade and DimAccount. DimTrade is a fact table that stores the information about the initiation and the completion or abortion of the trades, and DimAccount is a dimension table that stores the

---

**Algorithm 4.4 Update Data Placement Algorithm**

- **procedure** `UPDATEPLACEMENT(f_j, P, STAT)`
- 1: if `p_{j1} == 1 && p_{j2} == 1` then \{`f_j` is in both S-Store and Postgres\}
- 2: if \(B(EV, A, j) > 0\) then
- 3: \(p_{j1} \leftarrow 1, p_{j2} \leftarrow 0\) \{Evict from Postgres\}
- 4: else if `p_{j1} == 1` then \{`f_j` is in S-Store only\}
- 5: if \(B(CP, SA, j) > C_M(SA, j)\) then
- 6: \(p_{j1} \leftarrow 1, p_{j2} \leftarrow 1\) \{Copy to Postgres\}
- 7: else \{`f_j` is in Postgres only\}
- 8: if \(B(AS, j) > C_M(AS, j) \&\& \text{KEEPINSTORE}(f_j, P, STAT) == \text{True}\) then
- 9: if \(B(MV, AS, j) \leq B(CP, AS, j)\) then
- 10: \(p_{j1} \leftarrow 1, p_{j2} \leftarrow 1\) \{Copy to S-Store\}
- 11: else
- 12: \(p_{j1} \leftarrow 1, p_{j2} \leftarrow 0\) \{Move to S-Store\}
- 13: \(p_{j1} \leftarrow 1, p_{j2} \leftarrow 0\) \{Move to S-Store\}
information about the accounts. *DimTrade* has a foreign key on attribute *SK.AccountID* that references attribute *SK.AccountID* of table *DimAccount*.

Figure 4.6 shows the setting for our experiments. For the ingestion of *DimTrade*, there are five operations *OP1, ..., OP5*. Each operation refers to a procedure shown in Figure 4.3. For example, *OP4* refers to *SP4* in Figure 4.3, and is responsible for checking the foreign key that references attribute *SK.AccountID* of table *DimAccount*. For the ingestion of *DimAccount*, the operations are similar procedures. For example, *OP8* checks the foreign key on attribute *SK.CustomerID* of table *DimAccount* that references attribute *SK.CustomerID* of table *DimCustomer*.

Our experiments involve the ingestion of the *DimTrade* and *DimAccount* tables from their respective flat files. Each of these ingestion processes was modeled as its own dataflow graph consisting of multiple SQL statements. These SQL statements perform lookups on a variety of other tables to retrieve normalized foreign keys before inserting the finished tuple into a final table.
In the case of DimAccount, incoming tuples represent in-place updates to existing rows in the database. DimTrade tuples, on the other hand, are always inserts, but require a lookup on the DimAccount table to generate a foreign key. S-Store is configured as single-sited. Since there are no distributed transactions for this configuration, we chose to implement the ingestion of DimTrade and the update of DimAccount each in one stored procedure.

We generated heterogeneous workloads that contain changes to both DimAccount and DimTrade. In each experiment, the workload varies in terms of the percentage of operations that write or read from DimAccount. We partition DimAccount into ten fragments of equi-width. We randomly generate a sequence of in-place updates that follows Zipfian distribution. The reason for choosing Zipfian distribution is that it is the standard distribution measured from real-life OLTP workloads adopted by benchmarks such as YCSB [28]. We then mix the in-place updates with the data ingestion source from Trade.txt. We measure the average latency of each operation (in-place update or ingestion) for the heterogeneous workload.

For simplicity, most tables in this experiment are considered to be cached in S-Store. The DimTrade table is considered to be entirely located in S-Store. The DimAccount table, on the other hand, is primarily located in Postgres. In the following experiments, a percentage of DimAccount is either copied or moved to S-Store, depending on the scenario. For all of the experiments, we measure the average latency of each operation (ingestion or update) in the workload that includes necessary data migrations.
4.7.2 Results

Insert-intensive Workloads.

As shown in Figure 4.6, the ingestion of DimTrade contains five operations, with OP4 retrieving the account-id, broker-id and customer-id from table DimAccount. Executing this lookup process locally in S-Store (i.e., copying DimAccount from Postgres to S-Store) can generally improve the performance, but S-Store only has limited storage space (or cache as we call it in this paper). In this experiment, we study how the storage limit of S-Store affects the performance for lookups (to table DimAccount) during data ingestion (of table DimTrade). The workload we generate for this experiment contains only data ingestion to DimTrade and no update to DimAccount, i.e., this workload contains 100% inserts to DimTrade (thus 100% ETL-reads to DimTrade) and 0% writes to DimAccount.

Figure 4.7 demonstrates the benefit of copying tables to S-Store when there are lookups in the ETL tasks. As we clarified in Section 4.7.1, the y-axis represents the average latency of the ingestion, including necessary data migrations. When the cache size in S-Store is 0, for
each lookup to $\textit{DimAccount}$ during the ingestion to $\textit{DimTrade}$, the fragment of $\textit{DimAccount}$ that contains the key (account-id) must be migrated from Postgres to S-Store. Typically the migration incurs prohibitive cost. When the cache size increases, the fragments that have been migrated to S-Store can be stored locally for future lookups during the ingestion, reducing the number of migrations. We employed least recently used (LRU) to evict fragments from S-Store when the size of copied fragments exceeds the cache limit. When S-Store has a large enough storage and is able to cache all the fragments of $\textit{DimAccount}$ table that are required in the ingestion of $\textit{DimTrade}$, the latency of the ETL ingestion to $\textit{DimTrade}$ is minimized.

In this scenario, the average latency of the workload for moving is more expensive than copying for most cache sizes. The reason is that when the cache in S-Store is full, and a fragment required is not in the cache, a new migration must be issued. Copy only has to migrate the required fragment from Postgres to S-Store, while Move has an additional step to migrate the evicted fragment from S-Store back to Postgres. We observed that for this scenario, our online approach performs as good as Copy.

**Write-intensive Workloads.**

When migrating data from Postgres to S-Store, Move implies that there is always only one copy of the data in the database engines, while Copy implies that there are two copies of the data in the system: one in S-Store, and another in Postgres. The workload in this experiment contains 99% in-place updates to $\textit{DimAccount}$ and 1% ingestions to $\textit{DimTrade}$, generated as described in Section 4.7.1. In order to guarantee the transactional safety, the updates are executed synchronously in S-Store and Postgres. When an update is issued to S-Store, if the fragment of the data that this update accesses exists only in S-Store, the update is executed and finished in S-Store. If the fragment of the data exists in Postgres, S-Store will issue the update to Postgres for execution and stalls until Postgres finishes the execution.

S-Store is built on top of H-Store, a system that is designed to speed up transactionally safe updates, and hence for such operations, S-Store has a much lower latency compared to Postgres. Figure 4.8 shows that when the cache size increases, more fragments are migrated to
S-Store, and since Move only keeps one copy of a fragment in the system, moved fragments exist only in S-Store. Thus, there are no additional steps for updating the data in Postgres, which would increase the cost. Therefore, for a write-intensive workload where updates are the majority, the average latency decreases quickly when the cache size increases if we choose to move the data between S-Store and Postgres.

On the contrary, Copy keeps the data in both S-Store and Postgres. As we have explained, the cost of updates in Postgres dominates the synchronized update process, and thus the curve of the average latency for Copy does not change much when the cache size increases. We also notice that when the cache size is less than 10% of the size of the DimAccount table, Copy performs better than Move. It is not difficult to see that in such cases, a large amount of migrations are conducted because of cache misses (i.e., the required fragment is not in the cache). For each cache miss, Copy only has to migrate the required fragment from Postgres to S-Store (and delete the evicted fragment from S-Store), while Move must execute two migrations, one for the required fragment from Postgres to S-Store, another one for the evicted fragment from S-Store to Postgres.
We also observe that for a workload that simulates real-life OLTP workloads, the average latency of our online approach performs as good as Move when the case size is relatively large. When the case size decreases, comparing to LRU employed by Move, our online approach evicts fragments from S-Store based on the benefit and cost. This gives us the advantage of smart caching of the fragments that are more likely to be reused in future, and do not evict them too early.

**Heterogeneous Workloads.**

We have seen that for read-intensive workloads, Copy often has better average latency, and for write-intensive workloads, Move usually has better average latency. Here, we experiment with workloads that are heterogeneous. In this experiment, we generate a series of workloads for which insert operations (ingestion to *DimTrade*) make up from 1% to 100% of the total workload. We fix the cache size to 30% of *DimAccount*.

First, Figure 4.9 confirms our previous observation that when the percentage of reads is not big (< 60%) where the workload is dominated by writes (transactional updates to *DimAccount*), the latency for Move is lower than that for Copy, and when the workload is dominated by reads (ETL lookups to *DimAccount*), the latency for Copy is better than that for Move. Secondly, in our experiments, the cost of inserts is much cheaper than that of writes (in-place updates); thus, for the Copy scenario, the average latency of all inserts and writes to *DimAccount* in the workload decreases as the percentage of inserts in the workload increases. Thirdly, we notice that the average latency for Move increases when the percentage of insert in the workload increases. This is because for cache misses when the cache is full, Move is much more expensive than Copy, as we have explained above. The additional migration cost for Move offsets the benefit brought by cheaper insert. The figure shows that the curves for Copy and Move meet in a workload that contains about 20% of read operations in this setting as a confluence of the factors cited above. Lastly, we observe that our online approach performs better than Move and Copy for all workloads. This is due to two reasons: 1. Our online approach can dynamically move or copy data fragments based on cost and benefit. 2. The
Another advantage of our online approach is that the redesign procedure is not only triggered by necessary data migrations. Figure 4.10 shows a workload that contains 100% of updates. Since there is no insertions (and hence no ETL-reads), for Move and Copy there is no data migrations, because updates can be conducted in either S-Store or Postgres. In this case, all updates are executed in Postgres for both Move and Copy, and this is why the performance of Move and Copy are close to each other for this workload. On the contrary, since our online approach can periodically redesign the placement based on the workload, it identifies the update-intensive pattern, and migrates data accordingly. This greatly reduces the average latency of the updates.

**Takeaway Messages.**

We notice that the migration cost between database engines (S-Store and Postgres) is very expensive compared to the cost of local reads and writes in a workload. The migration cost is frequently not negligible during the execution of a mixed workload. For instance, although
local reads are much cheaper than writes in S-Store in our setting, an ETL read may incur a data migration from Postgres to S-Store, and it may increase the average latency for data ingestion by up to two orders of magnitude. This implies that for certain circumstances, migration cost could be the dominating cost for a workload, and minimizing this migration cost may be a good enough objective function for an approximated optimized design. For other circumstances, considering only migration cost is probably not enough. For instance, for a workload that contains only writes, it may make sense to migrate the data from Postgres to S-Store, so the transactional writes are executed faster in S-Store, even if it means paying the additional cost for migrating data between the database engines.

4.8 Conclusion

In this chapter we discussed an instance of the general problem that we defined in Chapter 2: the workload contains both read and write operations. For such a workload running in a distributed Polystore system, data placement and caching is a key factor for the performance. We
have shown that data ingestion in this setting presents some unique challenges. We applied and extended the solution in Chapter 2 to solve these challenges. Our solution involves an integration of a stream processing system with an analytics back-end provided by the BigDAWG polystore.

While caching and data placement are not new ideas, the context of a polystore changes their performance characteristics in such a way as to require a complete rethinking. In this work, we have considered copying results in the ingestion engine to make subsequent reads faster. Copying requires making or retaining a copy in the streaming engine or in the home storage system of the data. Any update to that data would have to be realized in all locations, making writes very expensive. To address this, we also allow moving the data, which simply moves the data to a new location (including perhaps the ingestion engine). This work has studied the problem of how to best match the workload to the appropriate moves and copies.

We solve the above challenge by developing a cost model to compare the effectiveness of multiple plans and accounts for the extreme expense that is incurred when migrating data from one system to another. Our algorithm based on the cost model adapts to changes in the workload by assuming that new operations are more similar to recent operations in the workload than they are to older operations. Our experiments show that when this is the case (as it is in a realistic workload adapted from TPC-DI, TPC-C, and YCSB), our cost model effectively optimizes the data placement and dramatically improves the workload performance.

In the next chapter, we will present a preliminary study of another instance of the problem defined in Chapter 2: for a read-only workload, combining materialization and data co-location, which co-locates data in a distributed shared-nothing system to speed up some common database operations such as joins, to improve the workload performance.
Chapter 5

Online Data Co-Location and Materialization

5.1 Introduction

In this chapter, we present a preliminary study of yet another instance of the general problem that is defined in Chapter 2. Similar to Chapter 3, we assume that workload $\mathcal{W}$ is read-only. We will study how materialization can be combined with another important physical design technique (data co-location) to improve the performance of analytical workloads running on shared-nothing distributed systems.

Different from a traditional analysis executed on a single node relational database management system (RDBMS), today’s analytical workloads pose a number of new challenges. The first challenge we consider is how to create a physical design for the massive data volume that is typically stored in a shared-nothing distributed system. Although physical design is a key factor for performance when executing a workload on large scale data, making an appropriate design is often a hard problem and thus is tackled independently from other problems correlated to physical design. In this work, we focus on data co-location, a physical design technique, that locates tables that have joinable attributes in the same node in a shared-nothing distributed system. When data is not co-located, in order to join two tables, data from the tables...
must be moved to the same node [9]. Such a process often incurs very high data transfer cost (remote I/O). It has been shown that a good data co-location design can avoid remote I/O and thus greatly improve the performance of operations such as joins [9, 16, 63]. Data co-location requires partitioning data, and it can be computed by either hash partitioning or range partitioning (a technique that we used in Chapter 3). In this work we consider hash partitioning, because it is more common and robust for join operations.

A second challenge is how to maximize the potential of materialization for analytical workloads. Materializing intermediate results of queries and reusing these results to answer future queries that share the same computation is an important technique to speed up query execution [27]. We have shown in Chapter 3 that DeepSea [37] can significantly improve performance by combining materialization with partitioning. However, DeepSea does not consider alternative query plans for a given query (a query is executed in the order that is provided by the user). An alternative approach is to enumerate execution plans with different join orders for each query in the workload and choose the plan that would bring the most benefit to the overall workload by materializing and reusing intermediate results, even though the plan is not optimal for the single query [73]. This approach expands the search space of the materialization process (including view selection and view matching). In order to make the solution tractable, the cost of creating a view and the benefit of reusing a view must be carefully modeled based on appropriate assumptions.

A third challenge is how to properly combine materialization with physical design (e.g., data co-location). As we will show below shortly, materialization and data co-location are correlated, and considering them together in the query optimizer has a better opportunity to improve performance than considering them independently. Materialized views are usually considered as augmented data structures derived from base tables. One way to combine materialization and data co-location in one unified design framework is to consider co-located base tables as materialized views, so that we can uniformly model the cost and benefit for view creation, selection and matching for the co-located tables and other materialized views. However, the combination of these two data structures is not trivial because base tables and materialized
views have different properties. For example, materialized views can be deleted from the system to make space for other materialized views, and they can always be re-computed from the base tables (or the existing materialized views) when required. Although we may delete a copy of a base table when we have multiple copies for the table (each copy uses a different physical design of the table), we must keep at least one copy of the base table at all times. Furthermore, when we consider materialization and data co-location together in the query optimizer, the search space of the design is again greatly expanded, and we must efficiently explore the search space without significantly sacrificing accuracy to develop a solution that is scalable and accurate enough.

A fourth challenge is how to automatically adapt a new design that considers materialization and data co-location jointly to the characteristics of the workload. It has been shown that access patterns of today’s analytical workloads change frequently [15, 37, 63]. It becomes important that materialization and data co-location are designed together in an online fashion, i.e., without an assumption of a prior knowledge of the workload, adjusting the design based on the history of the workload.

Both materialization and data co-location have been investigated for a long period as independent processes. We now explain why this may lead to suboptimal solutions.

**Materialization.** Materialization can effectively speed up query execution for real-life analytical workloads where different queries share the computation for subqueries. The query optimizer can choose an appropriate design for materialization. However, as we show in Example 5.1, when a traditional materialization technique uses a cost model to make decisions on creating materialized views and reusing the materialized views to rewrite a query plan, taking materialized views as only a logical representation of the results of the subqueries without considering the specific physical design of the base tables can lead to suboptimal plans.

**Example 5.1.** *Consider a database with two relations: Sales(s_item_sk, s_date, s_sales_price) and Item(i_item_sk, i_category_id). Consider a SQL query $Q_1$ in a workload $W$. $Q_1$ computes the total sales for each item category and sales date.*
SELECT s.s_date, i.i_category_id, sum(s_sales_price)
FROM sales s JOIN item i
ON s.s_item_sk = i.i_item_sk
GROUP BY s.s_date, i.i_category_id

Suppose we decide to materialize the join result of Sales and Item as a view \( V_1 \) \((s\_date, i\_category\_id, s\_sales\_price)\) during the execution of \( Q_1 \):

\[
\text{SELECT s.s_date, i.i_category_id, s\_sales\_price}
\text{FROM sales s JOIN item i}
\text{ON s.s_item_sk = i.i_item_sk}
\]

because we have obtained enough evidence that such a materialized view can benefit the execution of the workload \([37, 73]\). Assume the cost of generating this view \( V_1 \) is 100 units, and the cost of reusing \( V_1 \) to answer future queries is 50 units. In this case we do not consider the data layout of Sales and Item. However, if \( W \) is executed on a shared-nothing distributed system (e.g., Spark \([86]\)), we are able to co-locate Sales and Item on \( s\_item\_sk \) and \( i\_item\_sk \) during the execution of \( Q_1 \) \([36]\), i.e., all tuples in both tables that have the same values of \( s\_item\_sk \) and \( i\_item\_sk \) are re-located to the same node. The co-location of Sales and Item suggests that in the future the join between Sales and Item on attributes \( s\_item\_sk \) and \( i\_item\_sk \) can be executed on local nodes without data shuffling. Assume the cost of generating data co-location for Sales and Item is 90 units, and the cost of the local join between Sales and Item is 40 units. In this setting, the creation cost of the co-location plan is lower than that of materializing the join results of Sales and item, and the cost of reusing the co-located relations to answer the query is also lower than that of materialization. Comparing the cost and the potential benefit, we may prefer the data co-location plan over the materialization plan for this query.

**Data Co-Location.** Physical design is a key factor for query performance. Previous work \([9, 72, 85]\) has shown how data co-location can improve query performance. Since physical design is usually an expensive process, it can only be performed in an offline fashion, i.e., optimizing the design with respect to a representative workload, and re-optimizing the design once it is
confirmed that the access patterns of the workload have changed. An assumption of this approach is that the characteristics of a workload do not change frequently. However, research has shown that this assumption does not always hold [15, 37, 55]. We proposed an approach [36] that performs data co-location as a by-product of query processing, i.e., data co-location is automatically adapted to the workload characteristics. We adopt the same approach to perform data co-location in this chapter. Another severe problem of non-adaptive approaches is that physical design is usually tackled as an independent process, separated from other query optimization processes such as materialization. Example 5.2 shows why this can lead to a suboptimal design.

Example 5.2. Continuing the running example, assume that after \( Q_1 \) is executed, there is a SQL query \( Q_2 \) that computes the number of sales for each item category and sales date.

```sql
SELECT s.s_date, i.i_category_id,
       count(*)
FROM sales s JOIN item i
    ON s.s_item_sk = i.i_item_sk
GROUP BY s.s_date, i.i_category_id
```

**Assume the cost of materializing the join results of Sales and Item to \( V_2 \) \((s\_date, i\_category\_id)\)**

```sql
SELECT s.s_date, i.i_category_id
FROM sales s JOIN item i
    ON s.s_item_sk = i.i_item_sk;
```

is different from Example 5.1, i.e., creating the view \( V_2 \) has a cost of 80 units and reusing it has a cost of 30 units, but the cost of generating data co-location for Sales and Item is still 90 units, and the cost of the local join between Sales and Item is still 40 units. Then we may choose to materialize \( V_2 \), hoping that \( V_2 \) will be used to answer future queries, since the cost of materializing \( V_2 \) is lower than co-locating Sales and Item, and the cost of using \( V_2 \) to answer future queries is lower than joining Sales and Item locally.

**Challenges.** We propose to consider data co-location and materialization together in the query optimizer in an online fashion since they are inherently interconnected. There are many new
challenges facing our approach:

1. When and how should we co-locate what data for an online approach?

2. How do we deal with the interplay of data co-location and materialization in the query optimizer without assuming prior knowledge of the workload?

3. How do we make the design process inexpensive? We cannot utilize typical optimization techniques for data co-location design and materialization (e.g., integer programming or graph partitioning [63, 73]) in the query optimizer because such techniques are usually expensive. In order to make these techniques practical, we often have to make strong assumptions or heuristics to aggressively prune the search space of the design.

Contributions.

1. We propose to co-locate data as a byproduct of query processing.

2. We consider data co-location and materialization together in the query optimizer.

3. We design a cost-benefit model that works for data co-location and materialization in one unified framework. We developed an online algorithm based on this cost-benefit model to improve the workload performance.

5.2 Related Work

There are several lines of work related to our approach in this chapter: materialization; data placement for distributed systems; and online self-tuning for physical design. We have introduced the related work of materialization (answering queries using views and reusing intermediate query results) in Section 2.3. We discuss data placement and online self-tuning for physical design in this section.

Data Placement. Data placement is an essential factor for the performance of query execution in distributed systems [9, 72, 85]. Since the data placement of Google File System (GFS) [43]
and Hadoop [1] is random and thus there is no easy way to co-locate data in these systems, systems such as CoHadoop [40] and Hadoop++ [35] implement non-random data placement for Hadoop. However, how to choose between different data placement plans is not studied. How to partition and co-locate data in distributed systems has been studied extensively in recent years [30, 44, 59]. Kayyoor et al. [59] model co-location as a hypergraph problem, and minimize the average query span (the number of nodes needed to answer a query) for a workload. Schism [30] finds the minimum cut of a hypergraph to minimize the number of distributed transactions for a workload. Neither of the above approaches considers the communication cost (the size of the data moved from one node to another) in their objective functions. Golab et al. [44] reduces the co-location problem to a graph partitioning problem and minimizes the communication cost. Different from solving a hypergraph partitioning problem in Kayyoor et al. and Schism, graph partitioning proposed by Golab et al. is less general than hypergraph partitioning. However, Golab et al. takes the query optimizer that is responsible for partitioning and materialization as a black box, and only performs data co-locating on the execution plan returned from the query optimizer. In addition, unlike our approach, Golab et al. do not consider execution cost in the objective function. All the above work finds co-location plans for fixed workloads.

Automated Physical Design. We have discussed work related to automated physical design in Section 2.3. In this section we introduce additional research that are directly related to our work.

Unlike the techniques that optimize the design according to a known, fixed workload [71], techniques that are directly related to ours are online approaches [37, 73, 76] that do not assume a prior knowledge of the workload, but make design decisions based on the history of computations that have been executed so far. We note that traditional approaches for materialized view selection [10, 11, 17] focus on when and what views to be materialized, but they do not consider how different data placement strategies interact with the decisions for materialization. It has been proven that considering physical design in the query optimizer is beneficial [13, 75], but we are the first to consider the interaction between data co-location and materialization in
5.3 Preliminaries

In addition to the preliminaries that we have discussed in Section 2.4, we give the definitions for the terminologies that we use in the rest of this chapter. We assume there is a cluster with a certain number of nodes that store data. In the rest of this work, a partition of a relation is identified by a hash value, and the union of the partitions of all relations equals the domain. We use the word table and relation interchangeably. Similar to Section 3.3, we assume a database $D$ consists of $n$ relations $D = (R_1, R_2, \ldots, R_n)$, and each relation has a schema that consists of a name and a list of attributes $R_i (A_1, \ldots, A_l)$. Assume $R_i$ is an instance of $R_i$ and $R_i$ contains a certain number of tuples with the same arity as $R_i$.

**Definition 5.1 (Relation Partitioning).** A relation partitioning is a triplet $P = (R, A, h)$, where $R$ is an instance of relation $R$, $A$ is a set of attributes of $R$, and $h$ is a hash function over the attributes $A$.

The tuples in $R$ are partitioned on $A$ using $h$, and we use $R_c$ to denote a fragment of $R \{ t | t \in R \land h(t.A) = c \}$. $P(R, A, h)$ denotes the partitioning of $R$ on $A$ by $h(A)$: $P(R, A, h) = \{ R_c | \exists t \in R : h(t.A) = c \}$. Note that Definition 5.1 is different from the definition for partitioning in Chapter 3. Although they both partition data horizontally, in this chapter, the partitioning of a relation is hash partitioning, i.e., it is controlled by a hash function $h$ on a set of attributes $A$, while in Chapter 3, the partitioning is range partitioning, in which it is controlled by the ranges on an attribute of the selection of the queries in a workload.

Next we define relation co-location as a set of sets of $P(R, A, h)$ for the set of relation instances $R_1, \ldots, R_n$. For each relation instance $R_i$ there exists at least one partitioning $P(R_i, A_i, h_i)$.

**Definition 5.2 (Relation Co-Location).** Given two relation partitionings $P_1 = (R_1, A_1, h_1)$ and $P_2 = (R_2, A_2, h_2)$, we say $P = (P_1, P_2)$ co-locate $R_1$ and $R_2$ based on $A_1$ and $A_2$ if
<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.1: Two relations to be co-located

Figure 5.1: $P_1 = (R_1, A_1, h_1) \land P_2 = (R_2, A_2, h_2)$ for Table 5.1

$|A_1| = |A_2|$ and $h_1 = h_2$. Given a set $\mathbf{P}$ of relation partitionings, co-location is an equivalence relation over pairs of elements from $\{R_1, R_2, \ldots, R_n\}$.

**Example 5.3.** Assume there are two relations $R_1(A_1)$ and $R_2(A_2)$ and three nodes in a cluster. Table 5.1 shows the tuples in $R_1$ and $R_2$. Assume $h_1$ and $h_2$ are two hash functions that compute the answer of an attribute value modulo 3 (we use the symbol `%` to represent the modulo operation in this work), i.e., $h_1(A_1) = A_1 \% 3$ and $h_2(A_2) = A_2 \% 3$. Figure 5.1 represents a possible placement of the tuples on the nodes for $\mathbf{P} = \{(P_1, P_2)\}$ where $P_1 = (R_1, A_1, h_1)$ and $P_2 = (R_2, A_2, h_2)$.

**Example 5.4.** Continuing the running Example 5.3, assume there are two more relations in the database $(R_3(A_3, B_3)$ and $R_4(A_4, B_4))$, $h_3(A_3, B_3) = (A_3 + B_3) \% 3$, $h_4(A_4, B_4) = (A_4 + B_4) \% 3$, $P_3 = (R_3, (A_3, B_3), h_3)$, and $P_4 = (R_4, (A_4, B_4), h_4)$. Now the database has a co-location partitioning $\mathbf{P} = \{(P_1, P_2), (P_3, P_4)\}$.

Similar to Chapter 3, we adapt the definitions for workload (Definition 2.1) and workload history (Definition 2.2) by requiring that all queries in workload $W$ are read-only. The workload history indicates the characteristics of the workload, and our online design approach is based on the workload history.
We also adapt the definition of configuration (Definition 2.3) to specify that the data structure that we consider for physical design is relation co-location as follows:

**Definition 5.3 (Configuration).** Assume the domain of the relations is denoted by $D$ which contains $n$ relation instances $R_1, \ldots, R_n$ that are involved in a workload $W$. Assume the domain of the materialized views is denoted by $V$ containing $v$ materialized views. Assume $T = D \cup V$. A configuration $C_i$ is a pair of $(V, P)$ before the execution of $Q_i$ in workload $W$, where $P$ is the relation co-location of $T$.

Recall that there is a upperbound limit $S_{\text{max}}$ for the size of a configuration.

Since we consider multiple query plans for a single query in this chapter, we introduce a history pool for the query plans of all queries in the history.

**Definition 5.4 (Plan History Pool).** We use $\mathcal{PH}(i)$ to denote the enumerated query plans that are maintained for each query that has been executed up to query $Q_i$ in workload $W$.

Similar to Perez and Jermaine [73], a plan history pool $\mathcal{PH}(i)$ contains a sequence of query $(Q_1, Q_2, \ldots, Q_i)$, the enumerated plans for each query, and the cost attached to each subquery of the enumerated plans.

Most modern query optimizers can enumerate execution plans for a query. It is not uncommon that there exist hundreds of plans for a single query. In order to consider the workload history (including all plans for the queries in the workload history) in the design, one approach for the cost-based query optimizer is to pick only a few (dozens) best possible plans, compare them, and choose the plan with the least cost for execution [73]. We adopt the same approach in this chapter. How to pick only a small number of plans for consideration is beyond the scope of this thesis, and we do not discuss the details here. We assume that the query optimizer has generated a certain number of ($k$) potential plans for each query consisting of base relations only (i.e., no data co-location or materialization is used in a plan).
5.4 Problem Statement

We address the problem of Data Co-Location and Materialization in this work. We adapt the problem definition in Chapter 2 (Definition 2.3) to Data Co-Location and Materialization.

Similar to the previous chapters, we denote $\text{COST}_{\text{EXE}}(Q, C)$ to the cost of executing query $Q$ in configuration $C$, and $\text{COST}(C_i, C_{i+1})$ to the cost of creating materialized views and co-located relations from configuration $C_i$ to $C_{i+1}$.

**Definition 5.5 (Data Co-Location and Materialization).** Given workload $W$ consists of a set of $n$ Select-Project-Join-Aggregate (SPJA) queries $Q_1, ..., Q_n$ that is executed one query at a time, and all join operations in $Q_1, ..., Q_n$ are equijoins, find a sequence of configurations $C = C_1, ..., C_n$, such that the total execution cost of the queries in $W$

$$\text{COST}(W, C) = \sum_{i=1}^{n} \text{COST}_{\text{EXE}}(Q_i, C_i) + \sum_{i=1}^{n-1} \text{COST}(C_i, C_{i+1}) \quad (5.1)$$

is minimized subject to the following constraints:

1. $C_1 = (\emptyset, \emptyset)$. This refers to our assumption that at the beginning no views are materialized and each relation is partitioned in some way but no relations are co-located.

2. Assume $V_{\text{cand}}(Q_i)$ refers to the candidates of materialized views and $P_{\text{cand}}(Q_i)$ refers to the candidates of co-located relations that can be generated during the execution of $Q_i$ (introduced in Section 5.7). $C_{i+1} - C_i \subseteq V_{\text{cand}}(Q_i) \cup P_{\text{cand}}(Q_i), \forall 1 \leq i < n.$

5.5 Solution Overview

Since we consider multiple query plans for a single query in this chapter, we extend Algorithm 2.1 to solve the problem defined by Definition 5.5.

Algorithm 5.1 shows the overview of our system. When a query $Q$ is posed to the system, the query optimizer enumerates a set of query plans. Cost-based query optimizers rely on the plan enumerator to choose the best query plan for a single query. We assume the existence of this functionality in our system, and we do not go through the details of the implementation of the plan enumerator in this work.
After we have obtained a certain number of query plans, for each plan we start to logically
match the subqueries in the plan with materialized views and co-located tables in the view pool
in order to reduce the cost of executing the plan by reusing existing materialized views and
co-located tables (Subprocedure matchDesign, introduced in Section 5.8).

When the logical matching is finished, for each plan we obtain a set of candidates of new
materialized views and co-located tables in addition to the statistics attached to the plan and the
candidates (subprocedure determineCandidates, introduced in Section 5.7). Note that we only
consider those materialized views and co-located tables that would bring benefit by reusing
them to answer queries in the workload history, and where the accumulated benefit exceeds
the costs of creating them. The statistics for the materialized views and co-located tables are
then updated with the potential savings (subprocedure updateStatistics in Section 5.9.5). This
step is followed by updating the plan and adding the updated plan to a set for plan selection
(choosing which plan has the best performance for the workload).

We use our cost-benefit model to conduct the selection process (subprocedure selectRewriting
and selectDesign, introduced in Section 5.9), i.e., to pick a plan that improves the perfor-
mance of the workload history, based on the existing relations in the view pool and the new
materialized views and co-located relations generated by this plan. Once this plan is found, we
start to evict existing relations in the view pool until the pool size limit is met. This is followed
by the execution of the plan. After the plan is executed, the co-located relations partitioned
on different attributes are created, views are materialized and admitted to the view pool, the
statistics of the new admitted data structures is updated, and the process is finished.

As we have introduced at the beginning of this section, the main difference between Al-
gorithm 5.1 and Algorithm 2.1 is that we consider multiple query plans of a single query. In
order to accommodate this extension, for each potential plan of a query, we must compute the
statistics of all potential data structures candidates in all plans. Thus we have to introduce a
loop, and determine these data structure candidates first before we update their statistics. Note
that in Algorithm 2.1, subprocedure selectRewriting is much simpler because it only considers
one query plan and it does not depend on the candidates of the potential data structures. But in
Algorithm 5.1, we have to choose the best plan for a query based on the statistics of both the
existing data structures and potential new data structures.

One key difference between our solution and the former solutions such as Perez and Jer-
maine [73] is that we perform the physical design for materialization and data co-location
together in the query optimization process. While our approach considers the correlation of
materialization and data co-location, it brings new challenges such as a much larger search
space for the design, and we will address these challenges in the following sections.

5.5.1 Co-location

We perform data co-location during the computation of a join operation [36]. When there is
a join between two relations that are not co-located on the join attributes, in order to generate
the co-location of the relations, we first shuffle the data in the relations so that all tuples that
have the same values on the join attributes are on the same nodes. Depending on the cost
and benefit of generating and reusing the co-location of the relations, we may want to delete
the original relations that are not co-located on the join attributes of the current join operation
(introduced in Section 5.9). In the data co-location process, we do not push down selections
and projections. The selections, projections, and the join operation itself is performed after the
data co-location process is finished.
5.5.2 Assumptions

In order to develop a practical solution to the problem defined in Section 5.4, we make the following assumptions in this work:

1. The selections in each query in $W$ are in conjunctive normal form (CNF). For queries that are not in CNF, they are converted to CNF.

2. We only consider two-way joins in this work. Multi-way joins [62] and broadcast joins [2, 16] are different physical designs for data layout. We argue that these designs can be embedded into our approach with appropriate cost and benefit definitions.

3. We assume that the benefit and cost for materialized views and co-located relations are independent from each other. Considering the correlation of the benefits and costs between different materialized views and co-located relations can greatly increase the complexity of the problem, and we leave this to future work.

5.6 Example Workload

In this section we present an example workload that will be used to illustrate how our system works in the following sections. Assume that we have a set of relations defined by the schema of TPC-H [7], and our workload consists of a sequence of queries that are constructed from the following query templates:

1. Query template $Q_1$. Retrieves the shipping priority and the potential revenue of the orders that had not been shipped by a given date.

   \[
   \text{SELECT } l\text{-}orderkey, o\text{-}ordernum, o\text{-}shippriority, \\
   \text{sum}(l\text{-}extendedprice\times(l\text{-}l\text{-}discount)) \text{ as revenue} \\
   \text{FROM } customer, orders, lineitem \\
   \text{WHERE } c\text{-}custkey = o\text{-}custkey \text{ AND } l\text{-}orderkey = o\text{-}orderkey \\
   \text{ AND } c\text{-}mktsegment = [\text{SEGMENT}'] \\
   \text{ AND } o\text{-}ordernum < \text{date'}[\text{DATE}]' \text{ AND } l\text{-}shipdate > \text{date'}[\text{DATE}]' \\
   \text{GROUP BY } l\text{-}orderkey, o\text{-}ordernum, o\text{-}shippriority;
   \]
2. **Query template** \( Q_2 \). For a given quarter, obtain the customers that have returned their orders, and list the customers in descending order of the lost revenue.

```sql
SELECT c_custkey, c_name, c_acctbal, n_name, c_phone,
      sum(l_extendedprice*(1-l_discount)) as revenue
FROM customer, orders, lineitem, nation
WHERE c_custkey = o.custkey
  AND l_orderkey = o.orderkey
  AND c_nationkey = n_nationkey
  AND o_orderdate >= date'[DATE]'
  AND o_orderdate < date'[DATE]' + interval'3'month
  AND l_returnflag = 'R'
GROUP BY c_custkey, c_name, c_acctbal, n_name, c_phone
ORDER BY revenue desc;
```

3. **Query template** \( Q_3 \). Return the number of orders that have been shipped in a given year for the given ship modes.

```sql
SELECT l_shipmode, count(o_orderkey)
FROM orders, lineitem
WHERE l_orderkey = o.orderkey
  AND l_shipmode in ('[SHIPMODE1]', 'SHIPMODE2')
  AND l_shipdate >= date'[DATE]'
  AND l_shipdate < date'[DATE]' + interval'1'year
GROUP BY l_shipmode;
```

In the above query templates, \( Q_1 \) is adapted from the query template \( Q_3 \) in TPC-H, \( Q_2 \) is adapted from the query template \( Q_{10} \) in TPC-H, and \( Q_3 \) is adapted from the query template \( Q_{12} \) in TPC-H. We simplified the selections and removed the nested queries.

We assume that most query optimizers push down projections. We now show that there does not exist a common subquery among \( Q_1 \), \( Q_2 \), and \( Q_3 \), such that a materialized view can be created for such a subquery to speed up the execution of future queries generated from a different template. Let us investigate the join results between table `lineitem` and `orders` for each of the query templates. Consider the following three views:
1. $V_1$ is the materialized view of the subquery in $Q_1$ that computes the join between \textit{lineitem} and \textit{orders}:

\[
\text{SELECT l\_orderkey, o\_orderdate, o\_shippriority, l\_extendedprice, l\_discount, o\_custkey}
\text{FROM orders, lineitem}
\text{WHERE l\_orderkey = o\_orderkey;}
\]

2. $V_2$ is the materialized view of the subquery in $Q_2$ that computes the join between \textit{lineitem} and \textit{orders}:

\[
\text{SELECT l\_extendedprice, l\_discount, l\_returnflag, o\_orderdate, o\_custkey}
\text{FROM orders, lineitem}
\text{WHERE l\_orderkey = o\_orderkey;}
\]

3. $V_3$ is the materialized view of the subquery in $Q_3$ that computes the join between \textit{lineitem} and \textit{orders}:

\[
\text{SELECT l\_shipmode, o\_orderkey, l\_shipdate}
\text{FROM orders, lineitem}
\text{WHERE l\_orderkey = o\_orderkey;}
\]

We notice that $V_1$ does not contain $l\_returnflag$ and $l\_shipmode$ in its projection, although these two attributes are required to compute $Q_2$ and $Q_3$ respectively; $V_2$ does not contain $o\_shippriority$ and $l\_shipmode$ in its projection, and these two attributes are required to compute $Q_1$ and $Q_3$ respectively; $V_3$ does not contain either $l\_returnflag$ or $o\_shippriority$ in its projections, and these two attributes are required to compute $Q_1$ and $Q_2$ respectively. We can observe the same fact for other potential materialized views such as the join results of \textit{customer} and \textit{orders}. Therefore a potential materialized view of the join results of \textit{lineitem} and \textit{orders} created by a query generated by $Q_1$ cannot be reused to answer a query generated by $Q_2$ or $Q_3$, and it is the same for the potential materialized views of the join results of \textit{lineitem} and \textit{orders} created by a query generate by $Q_2$ or $Q_3$. 
Next we illustrate several possible query plans for $Q_1$, $Q_2$, and $Q_3$ that are enumerated by the query optimizer. In order to describe the query plans that consider data co-location, we introduce a new relational operator $\delta$ to represent the data co-location process as described in Section 5.5.1, and another relational operator $\triangleright$ to represent the local join operations that do not require remote data transfer as explained in Section 5.1.

Figure 5.2a shows the optimized plan $p_{1}^{opt}$ for $Q_1$. In $p_{1}^{opt}$, selections and projections are pushed down by the query optimizer. Figure 5.2b shows an alternative plan $p_{11}$ for $Q_1$. In $p_{11}$, $\delta$ represents the data co-location process. The selection, projection, and join are performed after the data co-location is finished.

Similarly, Figure 5.3a shows the optimized plan $p_{2}^{opt}$ for $Q_2$, Figure 5.3b shows an alternative plan $p_{21}$ for $Q_2$ that co-locates $\text{lineitem}$ and $\text{orders}$, Figure 5.3c shows an alternative plan $p_{22}$ for $Q_2$ that materializes the join results of $\text{lineitem}$ and $\text{orders}$, Figure 5.4a shows the optimized plan $p_{3}^{opt}$ for $Q_3$, and Figure 5.4b shows an alternative plan $p_{31}$ for $Q_3$. Plan $p_{21}$ and plan $p_{31}$ both perform the data co-location before the selection, projection, and join.

Figure 5.5 shows the query plans that reuse the co-located relations and materialized views to answer the query. Assume $\text{lineitem}$ and $\text{orders}$ are co-located on $l_{\text{orderkey}}$ and $o_{\text{orderkey}}$, and $\text{customer}$ and $\text{nation}$ are co-located on $c_{\text{nationkey}}$ and $n_{\text{nationkey}}$, i.e., $P = \{(P_1, P_2), \ldots\}$. 
(P_3, P_4) \) where \( P_1 = (\text{lineitem}, l\_orderkey, h_1), P_2 = (\text{orders}, o\_orderkey, h_1), P_3 = (\text{customer}, c\_nationkey, h_2), \) and \( P_4 = (\text{nation}, n\_nationkey, h_2) \). For simplicity, we use the symbols of the relations with a superscript of \( \delta \) to denote that the relations are co-located in the database. Figure 5.5a to 5.5c shows the potential plans that reuse co-located relations. Figure 5.5d shows a potential plan that reuses a materialized view \( V_2 \) (defined above), which can be created during the execution of the query plan \( p_{22} \), to answer a query generated by \( Q_2 \).

In Table 5.2, we show the cost of each of the above plans. The cost can be estimated using a query optimizer. We will use the cost to illustrate how we make a design depending on the characteristics of a workload.
We will consider the following workload $W_1$ and $W_2$ in the rest of this chapter. Each query in the workloads is instantiated by providing the parameters to the according query template ($Q_1$, $Q_2$, or $Q_3$).

1. $W_1$: ($q_{11}$, $q_{12}$, $q_{13}$, $q_{14}$, $q_{15}$, $q_{16}$)
   - $q_{11} = Q_1(’HOUSEHOLD’, ’2018-01-01’, ’2018-02-01’).
   - $q_{12} = Q_2(’2017-12-01’).
   - $q_{13} = Q_3(’2017-08-15’, ’AIR’, ’TRUCK’).
   - $q_{14} = Q_2(’2017-12-15’).
   - $q_{15} = Q_2(’2017-10-01’).
   - $q_{16} = Q_2(’2017-06-30’).

2. $W_2$: ($q_{21}$, $q_{22}$, $q_{23}$, $q_{24}$, $q_{25}$)
5.7 View and Co-Location Candidates

The creation of materialized views and co-located relations in our approach consists of two steps: (1) determine the candidates of materialized views and co-located relations; (2) filter out the candidates of materialized views and co-located relations that we have not gathered enough
evidence that the benefit of reusing them would exceed the cost of creating them; (3) after we obtain the subset of materialized views and co-located relations that are most beneficial, based on the current configuration and our statistics, create these materialized views and co-located relations (introduced in Section 5.9). If the view pool is full, we evict existing materialized views and co-located relations whose cost benefit ratio are not high enough. This completes the transfer from the old configuration to the new configuration. Next we introduce the statistics that we use to estimate the cost and benefit.

5.7.1 View and Co-Location Statistics

Table 5.3 shows the statistics that we maintain for the materialized views and co-located relations in our system. \( \text{Cost}(R) \) is the cost of computing a view and storing it. \( \text{Cost}(P) \) is the cost of co-locating a set of \( n \) relations \( R_1, \ldots, R_n \) and storing them, where \( P = (P_1, P_2, \ldots, P_n) \), \( P_i = (R_i, A_i, h) \), \( 1 \leq i \leq n \), \( A_i \) is an attribute of \( R_i \), and \( h \) is a hash function over \( A_i \). We maintain \( \text{Cost}(R) \) and \( \text{Cost}(P) \) because they are required when computing the benefit for \( R \) or \( P \). Since we have a set of enumerated query plans for each query, different from Chapter 3, whether we create a materialized view or co-located relations depends not only on the plan that creates them, but also on the optimal query plan (independent from other queries) that the cost-based query optimizer would choose. For instance, for an example workload shown in Section 5.6, let us consider a co-located relation candidate \( P = (P_1, P_2) \) where \( P_1 = (\text{lineitem}, I\text{.orderkey}, h) \) and \( P_2 = (\text{orders}, O\text{.orderkey}, h) \). There are different plans such as \( P_1^{\text{co-loc}}, P_2^{\text{co-loc}}, \) and \( P_3^{\text{co-loc}} \) that can create this co-located relation candidate. As we will show in Section 5.9.1, we will compute the difference between the plan that creates it and the optimal query plan (\( p_1^{\text{opt}}, p_2^{\text{opt}}, \) or \( p_3^{\text{opt}} \)) of the query. We will introduce how we compute the cost and benefit in more details in Section 5.9.

5.7.2 View Candidates

Like the Definition 3.6 in Chapter 3, our view candidates include joins, projections, and aggregations. Different from Definition 3.6, we also consider the views that use queries with
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(R)</td>
<td>Size of a relation R</td>
</tr>
<tr>
<td>TS(R)</td>
<td>Timestamps a view or a co-located relation can be used in $\mathcal{H}$</td>
</tr>
<tr>
<td>Cost(R)</td>
<td>Cost of creating a view</td>
</tr>
<tr>
<td>Cost(P)</td>
<td>Cost of co-locating a set of relations</td>
</tr>
<tr>
<td>B(R, $t_{\text{now}}$)</td>
<td>Decayed potential savings associated with each TS(R) in $\mathcal{H}$</td>
</tr>
</tbody>
</table>

Table 5.3: Statistics in the cost model

selections as our candidates, because in this chapter we isolate the problem of combining data co-location and materialization from the problem of adaptively partitioning materialized views. Combining these two problems is an interesting avenue for future work.

**Definition 5.6 (View Candidates).** For a query plan $p_{ij}$ of query $Q_i$ and view configuration $C_i$, the set $\mathcal{V}_{\text{cand}}(p_{ij})$ of view candidates for $p_{ij}$ contains all subqueries $Q_i'$ of $p_{ij}$ that fulfill the following conditions:

- $Q_i'$ is of the form $\gamma(q), q_1 \bowtie q_2, \pi(q_1)$, or $\sigma(q_1)$
- $Q_i'$ does not exist in configuration $C_i$

**5.7.3 Co-Location Candidates**

As we have introduced in the previous sections, we allow more than one data co-location design (one relation is co-located on different attributes with different relations) for the same relation or view in our system. For example, we allow the existence of co-locations $P_1 = (R, A, h_1)$ and $P_2 = (R, B, h_2)$, where $R(A, B)$ is partitioned on attribute $A$ and $B$ respectively by $h_1$ and $h_2$. For $P_1$, $R(A, B)$ is co-located with a relation $S(A, C)$ on attribute $A$ ($(P_1, P_3)$), where $P_3 = (S, A, h_1)$. For $P_2$, $R(A, B)$ is co-located with a relation $T(B, D)$ on attribute $B$ ($(P_2, P_4)$), where $P_4 = (T, B, h_2)$.

In this chapter, we allow data co-location among all views and relations. Next we define co-location candidates for a query plan.

**Definition 5.7 (Co-Location Candidates).** Assume a query plan $p_{ij}$ of query $Q_i$ has been rewritten by the logical matching process (introduced in Section 5.8) of the query optimizer. $p_{ij}$ consists of a set of relations, materialized views, and materialized view candidates $R_1, R_2, ..., R_n$
that are equijoin. For each pair of the relations \((R', R'')\) that are joined on a set of attributes \(A'\), if \((R', R'')\) are not co-located on the same set of attributes \(A'\), i.e., there does not exist \((P', P'')\) such that \(P' = (R', A', h')\) and \(P'' = (R'', A', h'')\), \((P', P'')\) is a co-location candidate for \(p_{ij}\). The set of co-location candidates \(\mathcal{P}_{\text{cand}}(p_{ij})\) for \(p_{ij}\) is the set of such sets of candidates.

**Example 5.5.** Continuing the running Example 5.4, consider query \(Q_i\):

```sql
SELECT * 
FROM R_1 JOIN R_2 ON R_1.A_1 = R_2.A_2 
JOIN R_3 ON R_2.A_2 = R_3.A_3
```

Assume that \(p_{ij}\) is a plan for \(Q_i\), and \(p_{ij}\) executes a left deep join. Before the execution of \(p_{ij}\), we have a co-location partitioning \(\mathcal{P} = \{(P_1, P_2), (P_3, P_4)\}\) where \(P_1 = (R_1, A_1, h_1)\), \(h_1(A_1) = A_1 \% 3\), \(P_2 = (R_2, A_2, h_2)\), \(h_2(A_2) = A_2 \% 3\), \(P_3 = (R_3, (A_3, B_3), h_3)\), \(h_3(A_3, B_3) = (A_3 + B_3) \% 3\), \(P_4 = (R_4, (A_4, B_4), h_4)\), and \(h_4(A_4, B_4) = (A_4 + B_4) \% 3\). For \(p_{ij}\), \(\mathcal{P}_{\text{cand}}(p_{ij}) = \{(P_1, P_2, P_5)\}\), where \(P_5 = (R_3, A_3, h_5)\) and \(h_5 = A_3 \% 3\).

### 5.8 View and Co-location Matching

After the query optimizer has enumerated a set of execution plans for query \(Q_i\), the next step is to match the subqueries of each plan to the materialized views and co-located relations that exist in the current configuration, so that the query plan has a better performance by reusing the materialized views and co-located relations. We also have to update the statistics of the materialized views and co-located relations. The next step, view selection, depends on the updated statistics for the views and co-located relations to determine which plan to choose for the current query for execution and what new materialized views and co-located relations should be generated.

As we have already mentioned in Chapter 3, logical matching between a set of views and a subquery is an undecidable problem in our setting of the problem. In addition to that, we introduce data co-location into the matching process, which makes the problem at least as hard. In practical systems, logical matching is often performed by checking a set of sufficient con-
ditions [8]. We adopt the same approach. Similar to the approach introduced in Section 3.8.1, we compute a signature for each view and the subquery and logically match them on the signatures. The signature contains a set of elements about the query, storing different information such as the relations accessed by the query, information about the join and selection predicates, etc. The signature abstracts certain syntactic information such as the join order.

Computing all pairs of the signatures between a set of views and a subquery can be inefficient, especially when there are a large number of materialized views and co-located relations in the view pool. Similar to Goldstein and Larson, we constructed an index structure called a filter tree for managing materialized views and co-located relations on different levels. Using this index structure to match a subquery with a set of materialized views and co-located relations can be performed efficiently. We refer the readers to Goldstein and Larson for details of how the signature of a materialized view or a subquery is computed, and how a filter tree is constructed.

Next we will discuss how we perform logical matching for materialized views and co-located tables together to improve the query plan.

### 5.8.1 View matching

Our filter tree consists of different types of nodes on different levels. In order to search a match between a subquery and a view, we compute the signature of the subquery and use this signature to search a match in the filter tree. We also adopt the single-view substitutes matching in Goldstein and Larson, i.e., we only try to reuse one materialized view to answer a subquery. When there are multiple matches, we choose the materialized view that gives the least cost after replacing the subquery with the view. Meanwhile, different parts of a query can use different views [46]. For example, let us consider a query generated from template $Q_2$ in Section 5.6. Assume the following view $V_{21}$ have been materialized, in addition to view $V_2$ that is introduced in Section 5.6: $V_{21}$ is the materialized view of the subquery in $Q_2$ that computes the join between $customer$ and $nation$: 
Figure 5.6: Query plan for $Q_2$ (reusing two views)

```sql
SELECT c_custkey, c_name, c_acctbal, n_name,
       c_phone, c_custkey
FROM customer, nation
WHERE c_nationkey = n_nationkey;
```

For a query that is generated from template $Q_2$, we may use both $V_2$ and $V_{21}$ to answer the query by joining them properly. This is shown in Figure 5.6.

### 5.8.2 Co-location matching

We extend the filter tree to support co-location matching by adding a set of nodes for co-located relations to the according level of the filter tree. The matching between a query and co-located relations is straightforward: when two relations are joined by the join attribute condition, the nodes for co-located relations can be used to answer the query by joining the co-located relations locally. The search keeps going down to lower levels to find potential materialized views for reusing. When we find the first reusable materialized view, we compare the cost of reusing the co-located tables and the cost of reusing the materialized view, and choose the one that costs less. After we have made a choice between reusing a materialized view and co-located relations to answer the query, we keep the search to lower levels in the filter tree to find potential materialized views that would costs less to answer the query. Our heuristics is that for a materialized view $V'$ that is a child of another materialized view $V$ in the filter tree, the cost of reusing $V'$ is less than the cost of reusing $V$. 
5.9 View and Co-Location Selection

We try to improve each enumerated plan of query \( Q_i \) by logical matching that uses the materialized views and co-located relations in the current configuration \( C_i \) to answer \( Q_i \). After the logical matching process is finished, we must choose a plan for the execution of \( Q_i \) (plan selection). We perform the plan selection by computing the cost and benefit of materializing the view candidates and co-location candidates for each plan, and choosing the plan that is likely to bring the largest benefit for the workload \( W \) based on the workload history \( H \). During the process of the plan selection, for each of the view and co-located relation candidates, we must decide whether we will materialize the view or co-locate the relations, and if we do, what existing views or co-located relations must be evicted from the current configuration \( C_i \) to make space for the new materialized views or co-located relations if the view pool is full.

In this section we describe how we compute the cost and benefit for a materialized view or co-located relations, how we compute the cost of a plan of a query, how we use the cost and benefit to make the selection on the views and co-located relations, and how we select a plan for execution.

5.9.1 Cost Model

In Chapter 3, we do not enumerate query plans; we only consider one query plan that is issued by the user (the left-deep tree plan of the query). The computation of the cost and benefit of a materialized view is rather simple in this case. The cost of the view is the cost for creating and storing the view based on the CPU time spent on generating this view when executing the query, and the benefit of the view is the difference between answering the query using the view and answering the query without using the view (Section 3.7.1).

In this chapter, the scenario becomes much more complicated because we must consider multiple query plans for each query in the workload. A plan that is suboptimal for a single query may be an optimal plan for the workload because it may be able to generate intermediate results that can be used to answer other queries in the workload. Since we consider multiple
 plans, we must define the cost and benefit of creating a materialized view or co-located relations differently from the definitions in Section 3.7.1.

**Definition 5.8 (Optimal Plan of a Query).** We use $p_{i}^{\text{opt}}$ to denote the execution plan that is optimal for a single query $Q_i$. $p_{i}^{\text{opt}}$ may use any materialized views or co-located relations in configuration $C_i$ to answer $Q_i$.

### 5.9.2 Calculating the cost of views and co-located relations

A query plan can generate more than one materialized view or co-located relations. For instance, in the example workload described in Chapter 5.6, $p_{21}$ in Figure 5.3b can co-locate relation `lineitem` and `orders`, `customer` and `nation`, in addition to creating potential materialized views of the intermediate results of the joins, selections, and aggregations. We use $\text{Cost}(V)$ to denote the cost of creating a view. $\text{Cost}(V)$ is the time spent on computing the view and storing it. Similarly, we use $\text{Cost}(P)$ to denote the cost of relation co-location $P$, and $\text{Cost}(P)$ is the time spent on computing the co-location and storing the co-located relations.

Since we consider multiple plans for each query, before we define the benefit of using a view $V$ to answer a query in the workload history $\mathcal{H}$, we must define the cost of the plans for the query, because the benefit of $V$ depends on which plan we choose for the query.

We assume that the optimizer can compute the total cost of a plan. For each query $Q_i$ in workload $\mathcal{W}$, assume the cost of the optimal plan $p_{i}^{\text{opt}}$ under configuration $C_i$ is $\text{Cost}(p_{i}^{\text{opt}}, C_i)$, and the cost of an alternative plan $p_{ij}$ that generates a set of materialized views and co-located relations under configuration $C_i$ is $\text{Cost}(p_{ij}, C_i)$, we use $\text{Cost}(p_{ij}, C_i)$ to denote the overhead cost of $p_{ij}$ under $C_i$, and compute $\text{Cost}(p_{ij}, C_i)$ as follows:

$$\text{Cost}(p_{ij}, C_i) = \text{Cost}(p_{ij}, C_i) - \text{Cost}(p_{i}^{\text{opt}}, C_i)$$

### 5.9.3 Calculating the (potential) benefit of views and co-located relations

We keep all the queries that have been executed in our history pool, alone with the enumerated query plans of each of the queries in the plan pool. Assume the materialized view candidates of
the current query is $V_{cand}$, and the co-located relation candidates is $P_{cand}$. In order to estimate how $V_{cand}$ and $P_{cand}$ could have been used to answer the queries in the history pool, we must examine the potential benefit of using a candidate $V \in V_{cand} \cup P_{cand}$ in a query plan $p_{ij}$ of query $Q_i$ in the history pool, assuming $V$ had been materialized at the time that $Q_i$ was issued.

We compute the new cost of each plan of $Q_i$ by using the views and co-located relations in $V = V_{cand} \cup P_{cand} \cup C_i$, and find the plan $p_i^{new}$ that improves the performance most over the current optimal plan $p_i^{opt}$. We use $B(p_{ij}, V)$ to denote the benefit of a plan $p_{ij}$, and calculate $B(p_{ij}, V)$ as follows:

$$B(p_{ij}, V) = \max(0, \text{Cost}(p_i^{opt}) - \text{Cost}(p_{ij}, V))$$

and $p_i^{new}$ is the plan such that

$$B(p_i^{new}, V) = \max_j (B(p_{ij}, V))$$

We use $V$ to denote the view candidates and the co-located relation candidates that are used in $p_i^{new}$. For $V \in V_i$, we calculate the potential benefit of $V$ for $Q_i$ as follows:

$$B(V, i) = B(p_i^{new}, V) \times \frac{\text{Cost}(V)}{\sum_{v \in V} \text{Cost}(v)}$$

After the potential benefit of a view or co-located relation candidate has been calculated, we are able to accumulate the potential benefit for all of the queries in the history pool and adjust the potential benefit properly so that the benefit of using the view candidate happened a long time ago is less than that of a recent reuse.

Similar to the techniques that we used in Chapter 3, we also use a decay function to reduce the effect of a using a view or co-located relations as more queries arrive. A decay function such as the one used in Chapter 3 will decrease the benefit of views in the past, so that the policy is biased to the more recent uses. This makes the views and co-located relations adapt to the characteristics of the workload and new queries. We define the accumulated benefit for
a materialized view or co-located relation as follows:

\[ B(V, t_{\text{now}}) = \sum_{p_{\text{new}} \text{ used } V} B(V, i) \cdot \text{DEC}(t_{\text{now}}, t) \]  

(5.6)

The definition of the decay function \( \text{DEC}(t_{\text{now}}, t) \) is similar to the definition in Section 3.7.1.

### 5.9.4 View selection and plan selection

Since there is a size limit \( \bar{S} \) for our view pool, for each plan of the current query \( Q_i \), we must determine what views and co-located relation candidates can remain in the pool after the execution, and what existing views and co-located relations must be evicted from the pool to make space for the new admitted views and co-located relations.

We can use a set of statistics including the accumulated benefit, the cost of the view, and the size of the view to determine whether the view should be materialized and stored in the view pool. We use the same equation in Chapter 2 to define the cost benefit ratio as follows:

\[ \Phi(V, t_{\text{now}}) = \frac{\text{Cost}(V) \cdot B(V, t_{\text{now}})}{S(V)} \]  

(5.7)

where \( S(V) \) is the size of the materialized view. The cost benefit ratio for co-located relations is defined similarly.

Our algorithm for view selection is a greedy algorithm based \( \Phi(V, t_{\text{now}}) \) for each view and co-location relation. After we compute \( \Phi(V, t_{\text{now}}) \) for each materialized view and co-located relation in the view pool, we sort them in ascending order. With the sorted \( \Phi(V, t_{\text{now}}) \) for each existing view and co-located relation in the pool and each view and co-located relation candidate for a plan of the current query, for each plan, we maintain a list for the views and co-located relation candidates that will be admitted to the pool and a list for the existing views and co-located relations that will be evicted based on \( \Phi(V, t_{\text{now}}) \). The next step is to choose a plan to execute \( Q_i \).

Assume there are \( n \) plans \( p_{i1}, \ldots, p_{in} \) for \( Q_i \). We use \( V_j \) to denote the union of the view candidates and co-location candidates that will be admitted to the view pool by plan \( p_{ij} \). We
compute the cost overhead of $p_{ij}$ by Equation 5.2. We choose the plan $p_{ij}$ that has the following property:

$$\max_j \sum_{v \in V_j} B(v, t_{now}) - \hat{Cost}(p_{ij}, C_i)$$  \hfill (5.8)

for execution. After the execution of the plan, we admit the view candidates and co-located relation candidates in the list that we maintain for the plan, and evict existing views and co-located relations in the pool to make space for the admitted views and co-located relations.

Note that one key difference between our approach and the approach of Perez and Jermaine [73] is that Perez and Jermaine perform view selection in two independent processes: (1) during the query optimization, a query plan that generates certain materialized views is picked for execution; and (2) after the execution of the query plan, if the total size of the materialized views in the view pool exceeds the size limit, they start to evict the materialized views that contribute less in the history pool. On the contrary, we perform view selection as one process during query optimization. Potential materialized views (or co-located tables) of the current query plan will not be materialized (or co-located) if their cost benefit ratio is not good enough to keep them stay in the view pool because the pool is full and the current materialized views and co-located tables have better cost benefit ratio. The benefit of our approach is twofold: (1) No additional temporary space is required to store the materialized views that are created during the execution of the current query if the pool is already full; (2) If the materialized views that are created by the current query cannot be kept in the materialized view pool (and thus would be evicted right after they are created), we may not choose this plan at all in the first place. Even if the plan is picked for execution because other plans are more expensive, the views will not be created, and thus we avoid the oscillation between the admission and the eviction of the views.

**Example 5.6.** Let us study workload $W_1$ in Section 5.6. We compute the potential accumulated benefit $B(P)$ of $P$ in $W_1$, where $P = (P_1, P_2)$, $P_1 = (\text{lineitem, l.orderkey, h}_1)$, and $P_2 = (\text{orders, o.orderkey, h}_1)$. To simplify the discussion, we assume that $P$ is the only co-located relation that is considered for reusing when executing a query in $W_1$, and the decay function is
disabled, i.e., $\text{DEC}(t_{\text{now}}, t) = 1$. We use the sequence number of a query to represent the time that it is executed.

For $q_{11}$, configuration $C_1 = (\emptyset, \emptyset)$, the potential benefit defined in Equation 5.5 $B(P, 1) = 0$ because $\text{Cost}(p_1^{\text{co-loc}}) > \text{Cost}(p_1^{\text{opt}})$. Therefore the accumulated benefit (defined in Equation 5.6) $\mathcal{B}(P, 1) = B(P, 1) = 0$. $P$ is filtered out as a co-located relations candidate in $q_{11}$. Plan $p_1^{\text{opt}}$ is executed for $q_{11}$, configuration $C_2 = (\emptyset, \emptyset)$.

For $q_{12}$, the potential benefit $B(P, 2) = \text{Cost}(p_2^{\text{opt}}) - \text{Cost}(p_2^{\text{co-loc}}) = 700 - 600 = 100$. The accumulated benefit $\mathcal{B}(P, 2) = \mathcal{B}(P, 1) + B(P, 2) = 100$. Since the overhead of creating $P$ by plan $p_{21}$ in configuration $C_2$ is $\text{Cost}(p_{21}, C_2) = 1250 - 700 = 550 > \mathcal{B}(P, 2)$, $P$ is filtered out as a co-located relations candidate in $q_{12}$. Plan $p_2^{\text{opt}}$ is executed for $q_{12}$, configuration $C_3 = (\emptyset, \emptyset)$.

For $q_{13}$, the potential benefit $B(P, 3) = \text{Cost}(p_3^{\text{opt}}) - \text{Cost}(p_3^{\text{co-loc}}) = 400 - 200 = 200$. The accumulated benefit $\mathcal{B}(P, 3) = \mathcal{B}(P, 2) + B(P, 3) = 300$. Since the overhead of creating $P$ by plan $p_{31}$ in configuration $C_3$ is $\text{Cost}(p_{31}, C_3) = 1000 - 400 = 600 > \mathcal{B}(P, 3)$, $P$ is filtered out as a co-located relations candidate in $q_{13}$. Plan $p_3^{\text{opt}}$ is executed for $q_{13}$, configuration $C_4 = (\emptyset, \emptyset)$.

For $q_{14}$, the potential benefit $B(P, 4) = \text{Cost}(p_2^{\text{opt}}) - \text{Cost}(p_2^{\text{co-loc}}) = 100$. The accumulated benefit $\mathcal{B}(P, 4) = \mathcal{B}(P, 3) + B(P, 4) = 400$. Since the overhead of creating $P$ by plan $p_{21}$ in configuration $C_4$ is $\text{Cost}(p_{21}, C_4) = 550 > \mathcal{B}(P, 4)$, $P$ is filtered out as a co-located relations candidate in $q_{14}$. Plan $p_2^{\text{opt}}$ is executed for $q_{14}$, configuration $C_5 = (\emptyset, \emptyset)$.

For $q_{15}$, the potential benefit $B(P, 5) = \text{Cost}(p_2^{\text{opt}}) - \text{Cost}(p_2^{\text{co-loc}}) = 100$. $B(P, 5) = \mathcal{B}(P, 4) + B(P, 5) = 500$. Since the overhead of creating $P$ by plan $p_{21}$ in configuration $C_5$ is $\text{Cost}(p_{21}, C_5) = 550 > \mathcal{B}(P, 5)$, $P$ is filtered out as a co-located relations candidate in $q_{15}$. Plan $p_2^{\text{opt}}$ is executed for $q_{15}$, configuration $C_6 = (\emptyset, \emptyset)$.

For $q_{16}$, the potential benefit $B(p, 6) = \text{Cost}(p_2^{\text{opt}}) - \text{Cost}(p_2^{\text{co-loc}}) = 100$. $B(P, 6) = \mathcal{B}(P, 5) + B(p, 6) = 600$. The overhead of creating $P$ by plan $p_{21}$ in configuration $C_6$ is $\text{Cost}(p_{21}, C_6) = 550 < \mathcal{B}(P, 6)$. Since the potential accumulated benefit of $P$ is higher than the overhead cost of plan $p_{21}$ for $q_{16}$, $P$ is kept as a co-located relation candidate for $q_{16}$. Plan
Now let us study the potential accumulated benefit $B(V_2)$ of materialized view $V_2$. Note that $V_2$ can only be used to answer queries generated from template $Q_2$ (we explained this in Section 5.6). Similarly, to simplify the discussion, we assume that $V_2$ is the only materialized view that is considered for reusing when executing a query in $W_1$, and the decay function is disabled, i.e., $\text{DEC}(t_{\text{now}}, t) = 1$.

For $q_{11}$, configuration $C_1 = (\emptyset, \emptyset)$, $V_2$ cannot be used to answer $q_{11}$, thus $B(V_2, 1) = 0$. Plan $p_1^{\text{opt}}$ is executed for $q_{11}$. Configuration $C_2 = (\emptyset, \emptyset)$.

For $q_{12}$, the potential benefit of $V_2$ is $B(V_2, 2) = \text{Cost}(p_2^{\text{mat}}) - \text{Cost}(p_2^{\text{opt}}) = 700 - 560 = 140$. The accumulated benefit $B(V_2, 2) = 140$. The overhead of creating $V_2$ in plan $p_{22}$ is $\text{Cost}(p_{22}, C_2) = 1350 - 700 = 650 > B(V_2, 2)$. $V_2$ is filtered out as a materialized view candidate in $q_{12}$. Plan $p_2^{\text{opt}}$ is executed for $q_{12}$. Configuration $C_3 = (\emptyset, \emptyset)$.

For $q_{13}$, $V_2$ cannot be used to answer $q_{13}$, thus $B(V_2, 3) = 0$. The accumulated benefit $B(V_2, 3) = B(V_2, 2) = 140$. Plan $p_3^{\text{opt}}$ is executed for $q_{13}$. Configuration $C_4 = (\emptyset, \emptyset)$.

For $q_{14}$, the potential benefit of $V_2$ is $B(V_2, 4) = \text{Cost}(p_2^{\text{mat}}) - \text{Cost}(p_2^{\text{opt}}) = 140$. The accumulated benefit $B(V_2, 4) = B(V_2, 3) + B(V_2, 4) = 280$. Since the overhead of creating $V_2$ in plan $p_{22}$ is $\text{Cost}(p_{22}, C_4) = 650 > B(V_2, 4)$, $V_2$ is filtered out as a materialized view candidate in $q_{14}$. Plan $p_2^{\text{opt}}$ is executed for $q_{14}$. Configuration $C_5 = (\emptyset, \emptyset)$.

For $q_{15}$, the potential benefit of $V_2$ is $B(V_2, 5) = \text{Cost}(p_2^{\text{mat}}) - \text{Cost}(p_2^{\text{opt}}) = 140$. The accumulated benefit $B(V_2, 5) = B(V_2, 4) + B(V_2, 5) = 420$. Since the overhead of creating $V_2$ in plan $p_{22}$ is $\text{Cost}(p_{22}, C_5) = 650 > B(V_2, 5)$, $V_2$ is filtered out as a materialized view candidate in $q_{15}$. Plan $p_2^{\text{opt}}$ is executed for $q_{15}$. Configuration $C_6 = (\emptyset, \emptyset)$.

For $q_{16}$, the potential benefit of $V_2$ is $B(V_2, 6) = \text{Cost}(p_2^{\text{mat}}) - \text{Cost}(p_2^{\text{opt}}) = 140$. The accumulated benefit $B(V_2, 6) = B(V_2, 5) + B(V_2, 6) = 560$. Since the overhead of creating $V_2$ in plan $p_{22}$ is $\text{Cost}(p_{22}, C_6) = 650 > B(V_2, 6)$, $V_2$ is filtered out as a materialized view candidate in $q_{16}$. Plan $p_2^{\text{opt}}$ is executed for $q_{16}$. Configuration $C_7 = (\emptyset, \emptyset)$.

Comparing the above two designs ($P$ and $V_2$), we compute that for workload $W_1$, co-locating relation $\text{lineitem}$ and $\text{orders}$ is a potential design when we execute $q_{16}$, while mate-
rializing $V_2$ is not considered as a potential design.

Example 5.7. Continuing with the running Example 5.6, let us study the co-located relation candidate $P$ for workload $W_2$. Assume that $P$ is the only co-located relation that is considered when executing a query in $W_2$. We know the overhead cost of $p_{21}$ is $\hat{\text{Cost}}(p_{21}, C_i) = 1250 - 700 = 550$, and the accumulated benefit for using $P$ to answer the queries in $W_2$, which are all generated from template $Q_2$, is $B(P, 5) = B(P, 1) \times 5 = (\text{Cost}(p_{21}^{\text{opt}}) - \text{Cost}(p_{21}^{\text{co-loc}})) \times 5 = (700 - 600) \times 5 = 500 < \hat{\text{Cost}}(p_{21}, C_i)$, where $1 \leq i \leq 6$, thus $P$ is filtered out as a co-located relation candidate for workload $W_2$.

Now let us study the materialized view candidate $V_2$ for $W_2$. Assume that $V_2$ is the only materialized view that is considered for reusing when executing a query in $W_2$. The overhead cost of $p_{22}$ is $\hat{\text{Cost}}(p_{22}, C_i) = 1350 - 700 = 650$. Similarly we can compute the potential accumulated benefit of $V_2$ when $q_{25}$ is processed by the query optimizer: $B(V_2, 5) = 140 \times 5 = 700 > \hat{\text{Cost}}(p_{22}, C_5)$. Thus for workload $W_2$, $V_2$ remains as a materialized view candidate for $q_{25}$, and $C_6$ could be $(V_2, \emptyset)$.

5.9.5 Updating cost and benefit

For each query plan, after we find the best rewritten plan by reusing a set of materialized views and co-located relations, we must update the statistics in Table 5.3 for the views and co-located relations in the set. The size of a view or co-located relations does not change; a new timestamp of using the view or co-located relations is added to the sequence of the timestamps; the accumulated benefit must be updated as described in Section 5.9.3.

When a materialized view or co-located relations are evicted from the system because the view pool is full, we must update the filter tree and the cost of the query plans that reuse these evicted materialized views or co-located relations to answer queries in the history pool. In order to simplify the computation, we use a heuristics that when a view that is used to answer the query is evicted, the cost of the query plan is reset to the plan that does not use any views or co-located relations to answer the query.
5.10 Preliminary Experimentation

In this section we introduce a preliminary experimentation result of our research. We compare our techniques to the following baselines:

1. No materialization or data co-location is performed.

2. Materialization is performed, but no data co-location is considered.

3. Data co-location is performed, but no materialization is considered.

Figure 5.7 is a preliminary result that shows the comparison of the performance of the above three baselines and our online design when there is no size limit of the view pool. We generate a 50GB TPC-H dataset with uniform distribution and store it on a cluster of five nodes. Each node contains 64 virtual cores and 132GB memory. We generate three workloads from the query templates of TPC-H. Each workload contains 100 queries. The first workload $W_1$ is generated from two query templates ($Q_3$ and $Q_5$ of TPC-H). The second workload $W_2$ is generated from four query templates ($Q_3$, $Q_5$, $Q_7$, and $Q_8$ of TPC-H). The third workload $W_3$ is generated from eight query templates ($Q_3$, $Q_5$, $Q_7$, $Q_8$, $Q_9$, $Q_{10}$, $Q_{12}$, and $Q_{18}$ of
TPC-H). All workloads are uniformly generated from the query templates. Materialization is implemented by the same approach of Perez and Jermaine [73]. To simplify the discussion, for data co-location, we only consider co-locating table \textit{lineitem} and \textit{orders} statically. The x-axis represents the workloads, the y-axis is the time spent to execute the workload. Since the total number of queries in each workload is the same, when there are fewer query templates ($W_1$ and $W_2$), it is more likely that we can rewrite the queries by using the materialized views and thus improve the overall performance. When there are more query templates contained in a workload ($W_3$), it is less likely that we can reuse the materialized views to answer the queries since the projections of the queries generated from different query templates are often non-overlapped. However, the query templates usually join the tables on the same join attributes, and thus it is more likely that co-located relations can be used to answer the queries. Therefore for workload $W_3$, data co-location is a better design than materialization. Figure 5.7 shows that materialization and data co-location each fits better for different scenarios. The preliminary result also shows that our cost-based online approach can dynamically choose appropriate designs for different scenarios, and thus improves the performance over the baselines.

5.11 Conclusions

In this chapter, we discussed how the general problem and its solution that we studied in Chapter 2 can be applied to the problem of combining materialization and data co-location to improve workload performance. Combining these two techniques leads to a much larger search space for the design. In order to efficiently explore the enlarged search space, we adapted the solution in Chapter 2, built a cost benefit model that unifies materialization and data co-location, and developed a greedy algorithm that based on the cost model to perform selection effectively. Our preliminary result shows that our techniques can effectively improve workload performance.
Chapter 6

Conclusions

In this thesis we studied a general problem of combining physical design and materialization to improve workload performance. We investigated several instances of this general problem: combining data partitioning and materialization for non-update workloads; data placement for hybrid workloads (including updates); and combining data co-location and materialization to improve the performance. All of our approaches work in an online fashion. We proved that our solution is broadly applicable to different instances of the general problem. In this chapter, we conclude our findings and experience obtained from developing these techniques, and propose several possible avenues for future study.

6.1 The Problem of Combining Physical Design and Materialization

Materialization and physical design have both been proved to be effective techniques to improve workload performance, but often they are considered as independent techniques in literature. This can lead to suboptimal design, while we presume that unifying materialization and physical design has the potential to significantly improve the performance. We believe the problem of the efficient combination of materialization and physical design is broadly applicable to today’s data processing systems. We formally defined this problem, and developed a solution that can be applied to different instances of the general problem.
6.2 Progressive Partitioning of Materialized Views

Materialization is an effective way to improve workload performance. However, for big data analysis where the data volume is high, it is common that materialized views can grow too large in order to be used effectively. Partitioning materialized views into fragments and using fragments to answer queries give us a fine-grained control on what to keep in the view pool and what to use to answer queries. We proposed to progressively partition materialized views based on the workload history, and consider fragment correlation when performing view and fragment selection. By using data and workloads that are modeled after a real-life system, we proved that our techniques can greatly improve the workload performance.

In the short term, there are several interesting ways in which we can improve DeepSea including considering how to merge consecutive fragments that are mostly accessed together and how to best partition views on multiple attributes. DeepSea contributes to a rich literature on adaptive, progressive physical design strategies. For a fixed memory overhead, DeepSea selects a set of partitioned views and fragments of views to optimize the query performance (or minimize the read overhead). In the future, we would like to consider updates and explore how our techniques could be used with different optimization goals (including minimizing update overhead). DeepSea relies on estimations to evaluate the cost and benefit of different candidate designs. We will study techniques that can improve the precision of the estimation, so that DeepSea can be more robust to estimation errors. DeepSea only considers the query plan with the lowest cost. This strategy can lead to missing opportunities if a plan is optimal for the workload, although it is suboptimal for the query. We will extend our work by adopting similar techniques (considering multiple query plans) described in Chapter 5.

6.3 Online Data Placement for Polystore Systems

While our work on the data placement design for a Polystore involves two systems that each run on a single machine, the data placement problem is further complicated once the individual systems within the Polystore are distributed across multiple machines. In this scenario, it is
not enough to consider the system-level location of the data, but also the location of the data on physical hardware. It is likely that distribution properties of individual systems must be considered when determining data locations at the Polystore level. Additionally, while we concurrently focus on only two systems for simplicity, the data placement problem becomes much more difficult in a configuration space of three or more systems. As the number of systems in a Polystore increases, so do the possible trade-offs considered when deciding how to place data.

In the future, we envision a system that dynamically moves and copies the data between database engines that are designed for different data models in response to a particular heterogeneous workload. In order to accomplish this, we will need several things.

- A tighter integration with the BigDAWG query optimizer that uses the cost model to choose the best query plan, which here amounts to picking the best copy and using it at the best point in a distributed query plan.

- A more robust cost model that considers correlations of data access.

- A more extensive experimentation framework for operations on different Polystore settings.

We are currently working on these research issues and intend to perform a thorough experimental evaluation using several benchmarks, including TPC-DI and an application furnished to us by a major credit card processing company.

Similar to Section 6.2, our system requires accurate estimations of the costs and benefits. In the future we would like to develop techniques that can estimate costs and benefits precisely by building more robust models.

### 6.4 Online Data Co-Location and Materialization

After studying the combination of materialization and partitioning, we studied how materialization can be combined with another important physical design technique, data co-location,
to improve performance for analytical workloads executed in shared-nothing distributed systems. We showed that the design for materialization and the design for data co-location are correlated, and we studied how to build a cost benefit model that unifies the design for materialization and data co-location in the query optimizer, such that the design can be performed efficiently in a significantly enlarged search space. We will work on the implementation of the system and the evaluation of our techniques comparing with state-of-the-art techniques in materialization and data co-location.

In this work, we assume that the plan enumerator is a black box. Obtaining more potential plans is important because we must compare the costs among them to select the best plan for the workload. Meanwhile, we cannot increase the number of generated plans without restrictions, because doing so greatly increases the search space of the plans and the latency of matching a query with the designs can increase dramatically. A closer integration of our cost model and the plan enumerator is required. We plan to work on this topic in the future.

An interesting avenue is to combine the techniques we studied in this chapter with the techniques that are developed in DeepSea. We notice that cost estimation in real-life systems is often a hard problem and can affect the precision of the current design techniques greatly. Another interesting avenue is to study how to estimate the cost more precisely when the queries in a workload are executed in parallel.
Bibliography


The ACM Special Interest Group on Management of Data (SIGMOD), pages 2077–2080, 2016.


