© Copyright by Utkarsh Hriday Srivastava 2006
All Rights Reserved
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

__________________________
Jennifer Widom   Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

__________________________
Rajeev Motwani

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

__________________________
Surajit Chaudhuri
Microsoft Research

Approved for the University Committee on Graduate Studies.
Abstract

Efficient query processing in any data management system typically relies on: (a) A profiling component that gathers statistics used to evaluate possible query execution plans, and (b) A planning component that picks the plan with the best predicted performance. For query processing in a range of new data management scenarios, e.g., query processing over data streams, and web services, traditional profiling and planning techniques developed for conventional relational database management systems are inadequate. This thesis develops several novel profiling and planning techniques to enable efficient query processing in these new scenarios.

When data is arriving rapidly in the form of streams, and many registered queries must be continuously executed over this data, system resources such as memory and processing power may be stretched to their limit. First, for a class of computation-intensive queries, we describe how system throughput can be increased by exploiting sharing of computation among the registered queries. Then, for a class of memory-intensive queries, we consider the case when system memory is insufficient for obtaining exact answers, and give techniques for maximizing result accuracy under the given memory constraints. We then consider a distributed setting such as that of a sensor network, and give techniques for deciding the placement of query operators at network nodes in order to minimize system-wide consumption of resources.

We then consider the scenario of web services, which have been emerging as a popular standard for sharing data and functionality among loosely-coupled systems. For queries involving multiple web services, we give algorithms for finding the optimal execution plan. Finally, we turn to the profiling component, and describe new techniques for gathering statistics by not looking at the data but only at the query results. Such a technique is required when data access for collecting statistics is infeasible, as for web services, but can also be useful in traditional databases.
Acknowledgments

First and foremost, I would like to thank my adviser Jennifer Widom. She has provided me with immense help and excellent guidance throughout my PhD. Not only did she steer me in the correct direction during the problem definition and solution stages, but she also provided invaluable help in presenting and disseminating my work to the research community. My presentation skills will always bear the mark of her guidance.

I thank Rajeev Motwani and Surajit Chaudhuri for taking time out of their schedules and agreeing to serve on my reading committee. I also thank Surajit for giving me the opportunity to intern twice at the Data Mining and Exploration Group at Microsoft Research. The first internship, under the mentorship of Gautam Das, was especially a great learning experience for me, and to some extent, molded the direction that I took during the rest of my PhD.

I also thank all the members of the Stanford Infolab, and especially the STREAM group, for helpful comments and discussions. A special thanks to Kamesh, for providing theory expertise and a most fruitful collaboration. I also thank Marianne and Andy for handling the administrative and computer issues respectively, and ensuring that the group runs smoothly.

During my stay at Stanford, I have enjoyed the company of a lot of close friends. Our interactions have ranged from fun activities to serious technical discussions, and I have learnt a lot from them. I thank, to name a few, Manjeera, Anish, Arvind, Dilys, Krishnaram, Mayur, Rajat, and Rajiv for this wonderful time. Last, but certainly not the least, I thank my parents for their continued affection and support throughout my PhD.
## Contents

Abstract ........................................ iv  
Acknowledgments ................................ v  

1 Introduction ................................ 1  
  1.1 Data Management Architecture .......... 1  
  1.2 New Scenarios and Requirements .......... 3  
    1.2.1 Data Streams .......................... 3  
    1.2.2 Web Services ......................... 5  
  1.3 Summary of Contributions and Thesis Outline .......... 6  
  1.4 Related Work ............................. 8  

2 Sharing Computation Among Continuous Queries .......... 11  
  2.1 Introduction ................................ 11  
  2.2 Preliminaries ................................ 14  
    2.2.1 Cost Model ............................ 16  
    2.2.2 Problem Statement .................... 18  
  2.3 Shared Execution Strategies ............... 18  
    2.3.1 Fixed Strategies ..................... 19  
    2.3.2 Adaptive Strategies ................. 20  
  2.4 Finding the Optimal Strategy .............. 24  
    2.4.1 Hardness and Set Cover ............... 24  
    2.4.2 Lower Bound on Cost ................ 25  
    2.4.3 Greedy Adaptive Strategy .......... 26  
    2.4.4 Analysis ............................. 27  

vii
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6.5 WSMS Cost</td>
<td>131</td>
</tr>
<tr>
<td>5.7 Related Work</td>
<td>133</td>
</tr>
<tr>
<td>5.7.1 Web Service Composition</td>
<td>133</td>
</tr>
<tr>
<td>5.7.2 Parallel and Distributed Query Processing</td>
<td>134</td>
</tr>
<tr>
<td>5.7.3 Data Integration and Mediators</td>
<td>134</td>
</tr>
<tr>
<td>5.7.4 Query Processing over Remote Sources</td>
<td>135</td>
</tr>
<tr>
<td>5.8 Conclusions</td>
<td>135</td>
</tr>
<tr>
<td>6 Building Histograms Without Data Access</td>
<td>137</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>138</td>
</tr>
<tr>
<td>6.2 ISOMER Architecture and Overview</td>
<td>142</td>
</tr>
<tr>
<td>6.3 Preliminaries</td>
<td>143</td>
</tr>
<tr>
<td>6.3.1 Notation</td>
<td>143</td>
</tr>
<tr>
<td>6.3.2 A Running Example</td>
<td>144</td>
</tr>
<tr>
<td>6.3.3 Histogram Structure in ISOMER</td>
<td>147</td>
</tr>
<tr>
<td>6.4 Incorporating Feedback</td>
<td>148</td>
</tr>
<tr>
<td>6.4.1 The Maximum-Entropy Principle</td>
<td>149</td>
</tr>
<tr>
<td>6.4.2 Formulation of the Optimization Problem</td>
<td>149</td>
</tr>
<tr>
<td>6.4.3 Solution of Optimization Problem</td>
<td>151</td>
</tr>
<tr>
<td>6.5 Dealing with Database Updates</td>
<td>153</td>
</tr>
<tr>
<td>6.5.1 A Linear-Programming Solution</td>
<td>154</td>
</tr>
<tr>
<td>6.5.2 Merging Histogram Buckets</td>
<td>155</td>
</tr>
<tr>
<td>6.6 Meeting a Space Budget</td>
<td>156</td>
</tr>
<tr>
<td>6.7 Summary of ISOMER</td>
<td>157</td>
</tr>
<tr>
<td>6.8 Experiments</td>
<td>159</td>
</tr>
<tr>
<td>6.8.1 Accuracy on Static Data</td>
<td>160</td>
</tr>
<tr>
<td>6.8.2 Accuracy on Changing Data</td>
<td>163</td>
</tr>
<tr>
<td>6.8.3 Running Time</td>
<td>165</td>
</tr>
<tr>
<td>6.9 Implementing ISOMER for a WSMS</td>
<td>167</td>
</tr>
<tr>
<td>6.10 Related Work</td>
<td>168</td>
</tr>
<tr>
<td>6.11 Conclusions</td>
<td>170</td>
</tr>
</tbody>
</table>
7 Conclusion

7.1 Thesis Overview .................................................. 171
7.2 Future Directions .................................................. 172
  7.2.1 Data Streams .................................................. 172
  7.2.2 Web Services .................................................. 174

Bibliography .......................................................... 176
List of Tables

1.1 Problem Space Considered in Thesis ........................................ 7
2.1 Variables in a Given Problem Instance ................................. 15
2.2 Input Parameters to Problem Instance Generator .................... 35
2.3 Performance over a Stream of Images ................................. 41
3.1 Problem Space for Approximation of Joins ............................ 49
List of Figures

1.1 Data Management Architecture .............................................. 2
2.1 Execution Algorithm for a Shared Execution Strategy .................... 16
2.2 Decision Tree for Example 2.3.5 ............................................. 21
2.3 Construction for Theorem 2.3.7 when $m = 9$ and $n = 3$ .................. 23
2.4 Greedy Adaptive Strategy .................................................... 26
2.5 Analyzed variant of Greedy .................................................. 28
2.6 Algorithm for Precomputing an Adaptive strategy ......................... 34
2.7 Effect of Filter Costs ........................................................ 37
2.8 Effect of Filter Selectivities ................................................. 38
2.9 Effect of Filter Participation ................................................ 39
2.10 Effect of Number of Filters ................................................ 40
2.11 Effect of Correlation Among Filters ....................................... 41
2.12 Effect of Filter Participation ................................................ 43
2.13 Effect of Number of Nodes Stored ........................................ 44
3.1 Sliding-window join with aggregation ....................................... 50
3.2 Sliding-window join execution ............................................... 51
3.3 Algorithm UNIFORM ....................................................... 62
3.4 Age curve for WAN experiment ............................................. 69
3.5 Recall obtained on WAN experiment ...................................... 70
3.6 Recall obtained on synthetic age-based data .............................. 72
3.7 Aggregation error vs. memory allocated, IC frequency-based model, $Z = 2,$ $\mu = \sigma = 100$ ................................................. 74
3.8 Aggregation error vs. population variance, UC frequency-based model, $Z = 2, \mu = 100, Memory=10\%$ ............................... 75
6.9  Error of 3D Histogram on make, model and color .................. 162
6.10 Error vs allowed number of buckets ................................. 163
6.11 Updating: uniform to correlated .................................. 164
6.12 Data updated from correlated to uniform .......................... 166
6.13 ISOMER running time vs number of buckets ...................... 167
Chapter 1

Introduction

This thesis addresses several challenges that arise in enabling efficient query processing in two new data management scenarios: one in which data is arriving continuously in the form of streams [18], and another in which data resides behind restricted, but highly standardized, interfaces called web services [126]. Both of these scenarios (described in detail in Section 1.2) have requirements that significantly depart from those of the traditional scenario, in which data typically resides on disks and is managed by a relational database management system.

To motivate our work, we first describe the typical data management architecture and advocate the same general design even for the new data management scenarios in consideration (Section 1.1). We then delve into the characteristics and requirements of our new scenarios and illustrate why numerous components of the traditional architecture need to be substantially revamped in order to support these scenarios (Section 1.2). In Section 1.3, we summarize our contributions towards addressing these new requirements, followed by an overview of related work in Section 1.4.

1.1 Data Management Architecture

The typical architecture of data management is as shown in Figure 1.1. In this architecture, the data management application does not interact with the data directly but through a data management system that resides between the client and the data. One of many salient aspects of this architecture is declarative querying: The user can submit queries to
CHAPTER 1. INTRODUCTION

the data management system in a high-level declarative language (e.g., SQL [31]), specifying “what” information is required, rather than “how” it has to be gathered. The system then attempts to execute the given query over the data as efficiently as possible.

A single query written in a declarative language generally has many possible methods of execution depending upon the way data is accessed, the order in which query operations are performed, and the algorithms that are used to perform the query operations. Each such method of execution is referred to as a query plan, and there might often be orders of magnitude difference between the efficiency of different plans for a given query. Thus, a data management system typically includes a planning component, also known as a query optimizer (see Figure 1.1), that is responsible for taking arbitrary queries in a declarative language and finding reasonably efficient, or “good”, plans for the queries. The planning component has an associated cost metric that is used to compare different plans and choose the least-cost or predicted most efficient one among them. The chosen plan is then executed by the execution engine.

In practice, it is usually not possible to distinguish between plans unless some knowledge is available about the data that the query is going to execute on. Thus, a data management system also includes a profiling component that examines the data beforehand and gathers certain useful summaries or statistics over the data. These statistics are then used
by the planning component as shown in Figure 1.1.

The immensely popular relational database management systems (RDBMSs) have the above overall architecture. A distinct advantage of this architecture is that due to declarative querying, the application can be quite simple, while much of the complexity is pushed into the data management system. This same advantage carries over directly to the new data management scenarios we are addressing. However, the techniques that have been developed for the planning and profiling components in the context of RDBMSs are inadequate when we begin to consider new data management scenarios, as discussed next.

1.2 New Scenarios and Requirements

In this section, we describe the new data management scenarios considered in this thesis, and show why the planning and profiling techniques developed for traditional RDBMSs need to be revisited in light of these new scenarios.

1.2.1 Data Streams

There is an emerging class of applications, such as monitoring of stock tickers, sensor data processing, network monitoring, and intrusion detection, in which data is arriving continuously in the form of streams and must be analyzed in real time. To support such applications, data stream management systems (DSMSs) have been proposed [86, 120, 131]. A DSMS supports declarative querying of data streams through continuous queries. As opposed to one-time queries in DBMSs (which terminate after being evaluated on the stored data), continuous queries are registered with the DSMS and are evaluated continuously on the incoming data stream until explicitly terminated by the user.

For example, in network monitoring, the headers of packets on the network might form a stream that is input to the DSMS. A network administrator might detect a denial of service attack on a target host by registering a query that continuously reports the total number of distinct IPs that have made a connection with the target host in the last few seconds, and checking whether this value is above a certain threshold.

The techniques required for both the planning and profiling components in a DSMS are substantially different from those in a DBMS. We next describe some of these differences.
Exploiting Sharing

Data streams often have high volumes of arriving data. Since continuous queries need to be evaluated on every new data arrival, and some of them (e.g. stream joins [18]) may also require some recent history of the stream to be stored, a DSMS may be constrained by computational power and/or memory.

Since continuous queries are long-running, substantial resource savings can be obtained by exploiting sharing of computation and/or state among different queries executing on the same streams. Thus, the planning component should be designed to perform sharing whenever possible. Although sharing is potentially beneficial even in a conventional DBMS, it is both harder to find sharing opportunities, and also usually has a lower payoff since with one-times queries, the set of queries that will be executing simultaneously is unpredictable.

Approximation under Insufficient Resources

As discussed above, a DSMS may be constrained in terms of resources such as computational power and memory. In some cases, it may simply not have sufficient resources to evaluate all queries exactly over all the incoming data, without getting farther and farther behind. Fortunately, many stream applications can tolerate limited imprecision in their query results. For example, in the denial-of-service query above, it might be acceptable to report the number of distinct IPs to within ±10% of its true value.

When resources are insufficient to produce exact answers and the application can tolerate approximate answers, the planning component can include the degree of inaccuracy produced by a plan as part of its cost. This cost metric is entirely different from the traditional cost metric of query running time or latency, and requires entirely new planning techniques.

Distributed Setting

A particularly important application of data stream processing occurs in sensor networks, which are characterized by power-constrained nodes and continuous distributed collection of data. Power-constrained nodes leads to an additional cost metric of interest for the query planner: the amount of power consumed on executing a plan, in addition to traditional metrics such as query latency.
Since communication in sensor networks is a significant power-consumer, huge savings can be obtained by doing in-network processing, i.e., by executing query operators at various network nodes rather than transmitting all the data to a DSMS and doing centralized processing. Thus, the planning component must decide where different query operators should be executed to minimize overall power consumed, a problem that did not arise in a traditional DBMS scenario.

**One-Pass Statistics**

In a DBMS, the profiling component can access the data in advance and gather statistics that are subsequently used to choose good plans for queries. However, for data streams, no advance access to the data is available and queries must be evaluated on data as soon as it arrives. The only solution is to use the past as an indicator of the future, i.e., to gather statistics over the recent history of the stream, and evaluate the query assuming that the immediately subsequent stream elements will have similar statistics. Moreover, as data streams in, only a limited amount can be stored due to memory constraints. Thus, the profiling component requires a redesign so that it can gather statistics over the recent history of the stream using limited memory, with the statistics being continuously updated as new data arrives, but with low processing overhead.

**1.2.2 Web Services**

*Web services* [126] are rapidly emerging as a popular standard for sharing data and functionality among loosely-coupled, heterogeneous systems. Many enterprises are moving towards a service-oriented architecture by putting their databases behind web services, thereby providing a well-documented, interoperable method of interacting with their data. Furthermore, data not stored in traditional databases also is being made available via web services. For example, a stream of stock quotes from the NASDAQ can be wrapped as a web service that given a stock symbol, provides the stock price. Despite the popularity of web services as a means for sharing data, there are currently very few solutions available that permit declarative querying over web services. The following planning and profiling considerations arise when dealing with web services.
Exploiting Parallelism

For a query involving multiple web services, significant speedup can be obtained by exploiting parallelism among the web services. Thus, the planning component must decide how to schedule web service calls to obtain maximum speedup. The cost metric in this case is the same as the traditional metric of query running time, but the techniques required to obtain maximum parallelism are quite different from those in a centralized DBMS. Techniques from parallel databases are also inapplicable for web services, since unlike nodes in a parallel database system that can be programmed, the web services have a fixed functionality that is typically not under the client’s control.

No Data Access for Statistics Collection

In a traditional DBMS, direct data access is available for collecting statistics. We noted that for data streams, although data access is available, statistics must be collected in one-pass using limited memory. For web services, the considerations are quite different from both the above scenarios. Web services usually do not allow direct access to the data behind them as required for collection of statistics. The only other way that information may be gained about the underlying data distribution is by observing the results of queries over the web services. Thus, the profiling component for web services must be redesigned such that statistics can be collected by looking at query results rather than by looking at the data.

1.3 Summary of Contributions and Thesis Outline

Given the two scenarios described in Section 1.2, and the fact that new techniques are required for both the planning and the profiling components, the overall problem space we address is shown in Table 1.1. The same table summarizes the contributions of this thesis and places them in the context of this problem space.

Sharing of Computation

In Chapter 2, the thesis considers sharing of computation among continuous queries where each query is a conjunction of filters and each filter may occur in multiple queries. When query filters are expensive to evaluate, significant savings in computation are achieved
1.3. SUMMARY OF CONTRIBUTIONS AND THESIS OUTLINE

<table>
<thead>
<tr>
<th>Data Streams</th>
<th>Planning Component</th>
<th>Profiling Component</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Exploiting sharing of computation (Chapter 2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Approximation under insufficient resources (Chapter 3)</td>
<td>Previous Work: aggregates [12, 46, 53], quantiles [12, 67, 69, 92], frequent items [32, 42, 82], histograms [65, 66, 70], and distinct values [58, 64]</td>
</tr>
<tr>
<td></td>
<td>3. Operator placement in sensor networks (Chapter 4)</td>
<td></td>
</tr>
<tr>
<td><strong>Web Services</strong></td>
<td>Minimizing query running time by exploiting parallelism (Chapter 5)</td>
<td>Gathering statistics without looking at data (Chapter 6)</td>
</tr>
</tbody>
</table>

Table 1.1: Problem Space Considered in Thesis

by sharing filter evaluations across queries. The interesting observation in this work is that when filters are expensive, the best strategy is not necessarily one in which the filters are evaluated in the same fixed order for all input. We show that finding the best shared execution strategy is NP-hard, and provide polylogarithmic approximations to the best strategy.

**Approximation Under Insufficient Resources**

Next, in Chapter 3, the thesis addresses the problem of computing approximate answers to a particular class of memory-intensive continuous queries when the available memory may be insufficient to keep the required state for exact answers. We consider *sliding-window joins* [18] over data streams which, for fully accurate answers, may require storing potentially huge windows on the streams being joined. The metric for approximation may be either to lose as few result tuples as possible, or alternatively, to provide the largest possible random sample of the join result, e.g., for estimating an aggregate over the join result. We show that neither approximation can be addressed effectively for arbitrary input streams, propose models that real-life streams adhere to, and address both approximation metrics under these proposed models.

**Operator Placement in Sensor Networks**

In Chapter 4, we consider data stream processing in a typical sensor network where data acquisition takes place at low-capability devices and the acquired data is then transmitted
through a hierarchy of nodes having progressively increasing computational power. Given a query with expensive filters, there is a tradeoff between evaluating filters at the lower nodes in the hierarchy thereby saving on communication of unfiltered data, and evaluating them at higher-up nodes thereby saving on computation at low-power nodes. We address this communication-computation tradeoff and give an operator-placement algorithm that minimizes overall power consumption.

Query Planning for Web Services

Next, in Chapter 5, we move to the web services scenario and propose a general-purpose Web Service Management System (WSMS) that enables querying multiple web services in a transparent and integrated fashion. We tackle a first basic WSMS problem: query optimization for Select-Project-Join queries spanning multiple web services. Our main result is an algorithm for arranging a query’s web service calls into a pipelined execution plan that optimally exploits parallelism among web services to minimize the query’s total running time. Surprisingly, the optimal plan can be found in polynomial time in contrast to traditional query optimization where the analogous problem is NP-hard.

Gathering Statistics Without Looking at Data

Finally, in Chapter 6, we consider the problem of statistics collection when direct data access may not be available, as in web services. To address this problem, we propose an alternative method of gathering statistics by looking at the query results rather than at the data. Our approach is principled, adapts readily to changes in the underlying data, and can also keep limited-size statistics by retaining only the most “important” information while discarding the rest.

1.4 Related Work

In this section, we give a high-level survey of work related to this thesis. More detailed description of work related to individual contributions is deferred to the corresponding chapters.
1.4. RELATED WORK

Data Stream Processing Systems

The contributions related to data streams mentioned in Section 1.3 were made in the context of the STREAM project [120] at Stanford. A general-purpose system has been developed as part of this project and is available for trial and also as source code [115]. The motivation for STREAM and the system details appear in several overview papers [18, 120]. During the course of the project practically every aspect of query processing over data streams has been considered: a language for querying data streams [11], adaptive query processing [22], scheduling of operators to minimize memory usage [17], approximation under insufficient resources [20], sharing of state among queries [13], collection of approximate statistics with limited memory [12, 21], and handling of application-defined timestamps [112].

There are several other stream processing systems that have been developed concurrently with STREAM. All these systems have similar goals, but significantly different approaches from STREAM. The Aurora [131] system, and the consequent version for a distributed setting, Borealis [1], differ in that they have taken a non-declarative approach towards querying streams. The TelegraphCQ [86] system takes a radically different adaptive approach towards query processing by using the eddy [15] as its fundamental operator. NiagaraCQ [38] is a system developed for continuous monitoring of data spread over the internet, but doesn’t explicitly handle data streams. Gigascope [43] and Tribeca [117] are stream processing systems geared towards the specific application scenario of network monitoring. Nile [6] is a system developed with particular focus on event detection and tracking.

Gathering Stream Statistics

This thesis does not touch upon the profiling component (i.e., statistics collection) for data streams, but it has been the topic of much previous work. Activity in this area was perhaps sparked off by the influential sketches approach for tracking frequency moments over a stream using limited space. Subsequently, there has been a lot of work on tracking various different kinds of statistics such as aggregates [12, 46, 53], quantiles [12, 67, 69, 92], variance and k-medians [21], frequent items [32, 42, 82, 91], histograms [65, 66, 70], and number of distinct values [58, 64] over data streams. The basic goal of all of these approaches is to track statistics using limited space and low update overhead for every stream arrival.
Web Services

Declarative solutions to query data-providing web services are beginning to appear [23], however currently they do not have any sophisticated optimization techniques built-in. Much of the work on web services has instead focussed on services that provide *functionality* instead of data. In such a scenario, web services are composed (or *orchestrated* [30]) into a workflow using a language such as BPEL4WS [27].

Our proposal of a WSMS is similar to the concept of mediators in data integration systems. However, the optimization concerns are different. We are concerned with obtaining maximum parallelism by appropriately scheduling web service calls that have a high latency. Data integration systems are concerned with minimizing the cost incurred at the data integration system itself.
Chapter 2

Sharing Computation Among Continuous Queries

As mentioned in Chapter 1, since data streams often have high volumes of arriving data, a DSMS may be constrained in terms of computational power and/or memory. With long-running continuous queries, one way to obtain resource savings is to exploit sharing of computation and/or state among different queries executing on the same streams. For example, if multiple queries have a common filtering condition, for every arriving stream tuple, computation can be shared by evaluating the filtering condition only once rather than once each for every query that the condition occurs in. In this chapter, we address the problem of finding the optimal way of sharing computation among a class of continuous queries that are conjunctions of (possibly) expensive filters.

2.1 Introduction

A particularly common class of continuous queries is one in which a number of filters are applied to the input stream to produce the output stream. With several such queries, there may be sharing, meaning the same filter may occur in multiple queries. In addition, the query filters may potentially be expensive to evaluate.

As an example, suppose several information analysts are all monitoring an incoming stream \( S \) of images. Each analyst identifies the images of interest to him by specifying a collection of filters \( F_1, \ldots, F_k \) that the image must satisfy. Thus, each analyst poses a continuous query of the form:
SELECT * FROM S WHERE $F_1 \land \ldots \land F_k$

In this scenario, filters will most likely be shared, as more than one analyst may be looking for certain characteristics. For example, one analyst might be interested in outdoor images (filter $F_1$) with at least five people in it (filter $F_2$), while another analyst might be interested in outdoor images (filter $F_1$ again) with at least one person clad in black (filter $F_3$).

Filters that detect patterns within an image are often expensive to evaluate, thereby motivating the need to share filter processing across queries to increase throughput. (For example, the filter in the OpenCV library [100] to detect number of faces, when running on a 1.8 GHz machine, takes an average of 0.5 seconds on an $800 \times 600$ image.) Numerous other applications such as network monitoring, video surveillance, monitoring of voice calls, and intrusion detection, also exhibit shared expensive predicates and have high throughput requirements (e.g., a video feed needs to be processed in real time).

This chapter focusses on how to best exploit sharing of computation among the queries in such a scenario. A naïve execution strategy is to evaluate each query independently on the incoming stream. However, since filters are shared, this approach can perform a significant amount of redundant work. A better alternative is to choose a shared execution strategy. In a shared strategy, filters are evaluated on each data item in some order dictated by the strategy, until all of the queries are resolved. (Since each query is a conjunction of filters, a query is resolved as soon as one of the query filters evaluates to false, or when all of the query filters evaluate to true.) In this way, filter evaluations are shared across all queries. Due to expensive filters, significant performance gains can be achieved by such sharing. The specific problem that we address in this chapter is that of finding the optimal shared execution strategy for any given collection of queries that share expensive filters.

Challenges

Finding the optimal shared execution strategy poses the following major challenges:

1. **Filter placement.** The decision whether a filter should be evaluated earlier or later in a shared strategy should be made by taking all of the following factors into account:
   - **Cost:** Filters with low cost should preferably be evaluated early, since they might resolve queries at lower cost.
   - **Selectivity:** The average fraction of incoming data items that satisfy a filter is referred to as the selectivity of that filter. Filters with lower selectivity should
2.1. INTRODUCTION

preferably be evaluated early, since they are more likely to resolve queries by evaluating to false.

- **Participation**: The number of queries that contain a given filter is referred to as the participation of that filter. Filters with higher participation should preferably be evaluated early, since they can decide the results of a larger number of queries.

For a given filter, these three factors may give contradictory suggestions for its placement, so we must devise a placement method that takes all the factors into account.

2. Execution Overhead. Executing a shared strategy incurs some amount of overhead, e.g., keeping track of which filters have been evaluated, and which queries have been resolved. This overhead is in addition to the cost of the filters evaluated by the strategy. Thus the overall choice of the optimal strategy must take into account the expected total cost of filters evaluated by a strategy as well as its execution overhead.

To address the above challenges, we first consider the space of possible shared execution strategies and outline two broad categories of strategies: fixed and adaptive. In a fixed strategy, the order in which filters are evaluated is predetermined and is the same for each data item on the stream. In an adaptive strategy, at any stage the next filter to be evaluated is chosen based on the results of the filters evaluated so far. Adaptive strategies (sometimes also called conditional plans [49]) have a higher execution overhead since at each step they incur the cost of choosing the next filter to evaluate. However, in terms of the expected total cost of filters evaluated, the best adaptive strategy is often superior to any fixed strategy. Specifically, we show problem instances where any fixed strategy has a cost $\Omega(\mu)$ times the cost of the best adaptive strategy, where $\mu$ is the maximum number of queries in which a filter is present (measuring the extent of sharing of filters across queries). Thus, as the extent of sharing and filter costs increase, the higher execution overhead of adaptive strategies is compensated for by the savings obtained in filter evaluation cost, thus motivating the need to design and analyze adaptive strategies.

We then consider the optimization problem of finding the least-cost adaptive strategy, where the cost of a strategy is the expected total cost of filters evaluated by it. A key idea behind our results is to model this optimization problem as a probabilistic version of the well-known set cover problem [63]. Based on the similarity to set cover, we first show a lower bound: it is NP-hard to find an adaptive strategy whose cost approximates the
optimal cost to within any factor smaller than \( \ln m \), where \( m \) is the number of queries. We then give a greedy adaptive strategy, and we show that its cost approximates the optimal cost to within a factor \( O(\log^2 m \log n) \), where \( m \) is the number of queries as before, and \( n \) is the number of filters. Our experiments indicate that the greedy strategy performs very well in practice.

Finally, we consider the problem of reducing the execution overhead for adaptive strategies. One method is to precompute and store which filter is to be evaluated next for each possible combination of outcomes of the filters evaluated so far. (Essentially, we would be materializing the “decision tree” corresponding to the adaptive strategy.) However, in general, this approach requires space exponential in the number of filters. We give an algorithm that takes into account the amount of space available to store the decision tree and decides which parts of the decision tree should be precomputed so that the execution overhead of the strategy is minimized.

Chapter Organization

The rest of this chapter is organized as follows. In Section 2.2, we formally define the problem of optimizing a collection of queries with conjunctions of shared filters. Then, in Section 2.3, we explore the space of shared execution strategies and show that when filters are expensive, the optimal strategy is adaptive. In Section 2.4, we show the hardness of finding (even approximating) the optimal adaptive strategy, and we give a greedy adaptive strategy that approximates the optimal strategy to within a factor \( O(\log^2 m \log n) \), where \( m \) is the number of queries and \( n \) is the number of filters. In Section 2.5, we show how the execution overhead of an adaptive strategy can be reduced by appropriate pre-computation. In Section 2.6, we give a thorough experimental evaluation showing that our techniques lead to a significant improvement in performance over more naïve techniques. Finally, we survey related work in Section 2.7, and conclude in Section 2.8.

2.2 Preliminaries

Consider an incoming stream \( S \) of data items. The data items may have any data model or type. Let there be a set of \( m \) queries \( Q = \{Q_1, \ldots, Q_m\} \) posed on stream \( S \). Each query \( Q_i \)
is a conjunction of filters on the items of $S$:

$$Q_i : \text{SELECT } * \text{ FROM } S \text{ WHERE } F^1_i \land \ldots \land F^k_i$$  \hspace{1cm} (2.1)

For the rest of the chapter, we denote the query $Q_i$ as the set of filters $\{F^1_i, \ldots, F^k_i\}$, omitting the implicit conjunction. Thus, $F \in Q$ denotes that filter $F$ occurs in query $Q$. Filters may be shared among queries. For example, $F^a_i = F^b_j$ denotes that the $a$th filter in query $Q_i$ is the same as the $b$th filter in query $Q_j$. Let there be $n$ distinct filters over all the queries in $Q$, denoted by the set $\mathcal{F} = \{F_1, \ldots, F_n\}$. For a given collection of queries and filters, we also define the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>maximum number of queries a filter is present in $Q$</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>maximum number of filters present in a query</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>number of $(F_i, Q_j)$ pairs where $F_i$ is present in $Q_j$</td>
</tr>
</tbody>
</table>

Table 2.1: Variables in a Given Problem Instance

Note that $\mu$ measures the extent of sharing of filters between queries. Also note that $\lambda \leq \min(m\kappa, n\mu)$.

A shared execution strategy $\mathcal{P}$ for the queries in $Q$ gives an order in which the filters in $\mathcal{F}$ should be evaluated on the items of stream $S$. Formally, a shared execution strategy in its most general form is defined as follows:

**Definition 2.2.1. (Shared Execution Strategy).** A shared execution strategy $\mathcal{P}$ for the set of queries $Q$ is a function that takes as input the set of filters evaluated so far and their results, and decides the next filter to be evaluated.

Strategy $\mathcal{P}$ in its general form is said to be adaptive. In the special case when $\mathcal{P}$ always evaluates filters in the same order for each data item, regardless of the results of the filters evaluated so far, $\mathcal{P}$ is said to be fixed. This classification is elaborated upon in Section 2.3.

The execution algorithm for a shared strategy $\mathcal{P}$ is as follows (see Figure 2.1). For each incoming item $s$ on stream $S$, we first initialize the status of every query in $Q$ to unresolved (Lines 2-3). At any stage, the next filter $F$ to be evaluated is chosen according to strategy $\mathcal{P}$, and $F$ is evaluated on item $s$ (Lines 5-6). If $F$ evaluates to false, then all queries that contain $F$ are resolved to false (Line 8). Otherwise, if $F$ evaluates to true, $F$ is removed from all the queries it is part of. Any query $Q$ that now becomes empty (i.e., $F$ was the only remaining filter in $Q$), is resolved to true (Lines 10-11). This process is continued until
Algorithm $ExecuteStrategy(\mathcal{P})$

$\mathcal{P}$: Shared execution strategy for the set of queries $\mathcal{Q}$

1. for each data item $s$ on stream $S$
2. for each query $Q \in \mathcal{Q}$ /* Initialization */
3. \hspace{1cm} status($Q$) ← unresolved, numFiltersLeft($Q$) ← $|Q|$
4. while (status($Q$) = unresolved for some $Q$)
5. \hspace{1cm} choose next filter $F$ to be evaluated according to $\mathcal{P}$
6. \hspace{1cm} evaluate $F$ on $s$
7. \hspace{1cm} if ($F$ evaluates to false)
8. \hspace{2cm} for each $Q$ where $F \in Q$ status($Q$) ← false
9. \hspace{1cm} else
10. \hspace{2cm} for each $Q$ where $F \in Q$ numFiltersLeft($Q$) --
11. \hspace{1cm} if (numFiltersLeft($Q$) = 0) then status($Q$) ← true

Figure 2.1: Execution Algorithm for a Shared Execution Strategy

all queries in $\mathcal{Q}$ are resolved. We assume the set of queries $\mathcal{Q}$ is indexed on filters, so that for any filter $F$, the queries that contain $F$ can be determined efficiently (so that Lines 8 and 11 can be executed efficiently).

Next we describe our cost model for shared execution strategies (Section 2.2.1), then formally define the problem of optimization of queries with shared expensive filters (Section 2.2.2).

### 2.2.1 Cost Model

In order to compare different shared execution strategies and to choose the best one among them, we need to associate a cost expression with every strategy. The execution cost of a shared strategy consists of two major components:

1. **Cost of filters evaluated.** This is the total cost incurred in Line 6 of Algorithm $ExecuteStrategy$ in Figure 2.1.

2. **Execution overhead.** This is the cost incurred in the rest of the algorithm in Figure 2.1, i.e., excluding Line 6. Execution overhead consists of two parts—bookkeeping cost (such as keeping track of the number of filters remaining in each query), and the cost of adaptivity, i.e., the cost incurred by $\mathcal{P}$ in deciding the next
filter to be evaluated (Line 5). The cost of adaptivity depends on the specific plan $\mathcal{P}$, but the bookkeeping cost is independent of $\mathcal{P}$ and (for any tuple) is at most $O(1)$ per filter present in a query, making it $O(\lambda)$ overall.

We are focusing on applications with expensive filters (e.g., detecting patterns in images, and others mentioned in Section 2.1), and thus the first component of execution cost typically dominates the second. We primarily focus on filter evaluation cost and address the problem of designing execution strategies that minimize the expected total cost of filters evaluated. In practice, it is also important to keep the execution overhead low, and we give techniques for doing so in Section 2.5.

To arrive at an expression for the expected total cost of filters evaluated by a strategy, as in much of previous work [22, 36, 76], we assume that for each filter $F_i \in \mathcal{F}$, the following two quantities are known to the planning component (in practice, these quantities can be tracked by the profiling component):

- **Cost**: The average per-item processing time (or intuitively, the cost) of filter $F_i$ is denoted by $c_i$.

- **Selectivity**: The average fraction of data items that satisfy filter $F_i$ is referred to as the selectivity of filter $F_i$, and is denoted by $s_i$. The selectivity $s_i$ can also be interpreted as the probability $\Pr[F_i = \text{true}]$, where the probability is taken over the distribution of input data items.

For simplicity of presentation and analysis, we assume independent filters in the remaining discussion, i.e., the selectivity of a filter does not depend on the filters already evaluated. Note that our algorithms are more general, and work even for correlated filters (as demonstrated by our experiments in Section 2.6.2.5); we mention the simple modifications needed wherever appropriate.

Consider a shared execution strategy $\mathcal{P}$. Let $e_i$ be the probability that $\mathcal{P}$ will evaluate filter $F_i$ on an incoming item $s$. In general, $e_i$ may be less than 1 because all the queries might get resolved before $F_i$ is evaluated. Then the expected cost of filters evaluated by strategy $\mathcal{P}$ is given by:

$$\text{cost}(\mathcal{P}) = \sum_{i=1}^{n} e_i \cdot c_i$$  \hspace{1cm} (2.2)

The exact expression for $e_i$ depends on the specific type of strategy $\mathcal{P}$ and is given in Section 2.3. $e_i$ can be written in terms of selectivities of filters as shown by the following
Example 2.2.2. Let there be two queries \( Q_1 = \{ F_1, F_2 \} \) and \( Q_2 = \{ F_2, F_3 \} \). Let \( \mathcal{P} \) be a fixed execution strategy that always evaluates the filters in the order \( F_1, F_3, F_2 \). According to strategy \( \mathcal{P} \), filters \( F_1 \) and \( F_3 \) will always need to be evaluated. Thus \( e_1 = e_3 = 1 \). However, filter \( F_2 \) will need to be evaluated only if at least one of \( F_1 \) or \( F_3 \) evaluates to true. Since filters are independent, the probability that \( F_2 \) will need to be evaluated (i.e., \( \Pr[F_1 = \text{true} \lor F_3 = \text{true}] \)) is \( e_3 = s_1 + s_3 - s_1 s_3 \). Thus the cost of strategy \( \mathcal{P} \) according to (2.2) is:

\[
\text{cost}(\mathcal{P}) = c_1 + c_3 + (s_1 + s_3 - s_1 s_3)c_2
\]

2.2.2 Problem Statement

The problem of optimizing queries with shared expensive filters can be defined as follows:

Definition 2.2.3. (Optimization of Queries with Shared Expensive Filters). Given a set of queries \( Q \) of the form (2.1), find a shared execution strategy \( \mathcal{P} \) (Definition 2.2.1) such that \( \text{cost}(\mathcal{P}) \) given by (2.2) is minimized.

The problem of finding the best adaptive strategy is NP-hard (Theorem 2.4.2). We therefore focus on designing approximation algorithms. We give the standard definition of approximation ratio that is used to measure the quality of an approximation algorithm.

Definition 2.2.4 (Approximation Ratio). An algorithm \( A \) has an approximation ratio \( k \) (or is a \( k \)-approximation) if for all possible instances of the problem, \( A \) is guaranteed to result in a solution whose cost is at most \( k \) times the cost of the optimal solution.

2.3 Shared Execution Strategies

Recall Definition 2.2.1 of a shared execution strategy. In general, a strategy \( \mathcal{P} \) decides the next filter to be evaluated based on the results of the filters evaluated so far. Such a strategy is referred to as an adaptive strategy. However, a simple special case is when \( \mathcal{P} \) evaluates filters in a fixed order, independent of the results of the filters evaluated so far. In this case, \( \mathcal{P} \) is referred to as a fixed strategy. We study these two types of strategies in the following subsections.
2.3.1 Fixed Strategies

**Definition 2.3.1.** (Fixed Strategy). A fixed strategy is a shared execution strategy that evaluates the filters in \( \mathcal{F} \) on any data item in a fixed order (say \( F_1, \ldots, F_n \) without loss of generality). No redundant work is done: if the evaluation of \( F_1, \ldots, F_{i-1} \) resolves all queries that contain \( F_i \), the evaluation of \( F_i \) is skipped.

We first show how to calculate the cost of a fixed strategy \( \mathcal{P} \). For this, we need to find the probability \( e_i \) that \( \mathcal{P} \) evaluates \( F_i \) (recall (2.2)). \( F_i \) will be evaluated only if the evaluation of \( F_1, \ldots, F_{i-1} \) is not sufficient to resolve all queries that \( F_i \) is part of. Unfortunately, there is no simple closed form for the probability \( e_i \). However, \( e_i \) is simple to calculate as follows. Consider each possible result \( v \) of the evaluation of \( F_1, \ldots, F_{i-1} \) (there are \( 2^{i-1} \) possible values for \( v \)). Calculate the probability \( p(v) \) of \( v \) by using independence of filters and that \( \Pr[F_k = \text{true}] = s_k \) for any \( k \). Let \( b(v) \) be the indicator variable denoting whether any query containing \( F_i \) remains unresolved if the result of evaluating \( F_1, \ldots, F_{i-1} \) is \( v \). Then \( e_i \) is given by

\[
e_i = \sum_v p(v) \cdot b(v)
\]

**(Example 2.3.2.**) We redo Example 2.2.2 using (2.3). Recall the fixed strategy \( \mathcal{P} = F_1, F_3, F_2 \). Since \( F_1 \) is the first filter in \( \mathcal{P} \), \( e_1 = 1 \). Next, since \( F_3 \) needs to be evaluated regardless of the result of evaluating \( F_1 \), we have \( e_3 = 1 \). To calculate \( e_2 \), we consider all possible results of evaluating \( F_1, F_3 \).

<table>
<thead>
<tr>
<th>( v )</th>
<th>( p(v) )</th>
<th>( b(v) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_1 = \text{false}, F_3 = \text{false} )</td>
<td>( (1-s_1)(1-s_3) )</td>
<td>0</td>
</tr>
<tr>
<td>( F_1 = \text{false}, F_3 = \text{true} )</td>
<td>( (1-s_1)s_3 )</td>
<td>1</td>
</tr>
<tr>
<td>( F_1 = \text{true}, F_3 = \text{false} )</td>
<td>( s_1(1-s_3) )</td>
<td>1</td>
</tr>
<tr>
<td>( F_1 = \text{true}, F_3 = \text{true} )</td>
<td>( s_1s_3 )</td>
<td>1</td>
</tr>
</tbody>
</table>

The last 3 rows in the table above have \( b(v) = 1 \) since at least one of \( Q_1 \) and \( Q_2 \) (both of which contain \( F_2 \)) remain unresolved. Thus \( e_2 = (1-s_1)s_3 + s_1(1-s_3) + s_1s_3 = s_1 + s_3 - s_1s_3 \) as in Example 2.2.2.

Fixed strategies are of interest because it is known from previous work that for the special case of a single query (i.e., \( m = 1 \)), the optimal strategy is a fixed one, and is given by the following theorem\(^1\).

\(^1\)The problem is NP-hard when filter selectivities are correlated; however, the same greedy ordering modified to use conditional selectivities yields an approximation ratio of 4.
Theorem 2.3.3. [76] For a single query that is a conjunction of filters $F_1, \ldots, F_n$, it is optimal to evaluate the filters in increasing order of rank, where $\text{rank}(F_i) = c_i/(1 - s_i)$.

Now consider the strategy $\text{FixedGreedy}$ that is a naïve extension of the above theorem to multiple queries. $\text{FixedGreedy}$ processes the queries sequentially in arbitrary order. For each query, the filters are evaluated in the order given by the single-query greedy algorithm (Theorem 2.3.3). We now show a performance bound for $\text{FixedGreedy}$.

Theorem 2.3.4. $\text{FixedGreedy}$ is a $\mu$-approximation (recall definition of $\mu$ from Table 2.1).

Proof. Suppose the optimal strategy interleaves evaluation of filters from different queries. However, for any single query, the expected cost of the filters evaluated by the optimal strategy for resolving that query is at least the cost spent by $\text{FixedGreedy}$ (since $\text{FixedGreedy}$ is optimal for any single query). Since each filter appears in at most $\mu$ queries, the cost of $\text{FixedGreedy}$ can be at most $\mu$ times that of the optimal strategy.

The above theorem shows that the performance of $\text{FixedGreedy}$ deteriorates as the extent of sharing, $\mu$ increases. In Theorem 2.3.7, we will show that this approximation ratio of $O(\mu)$ is the best possible for any fixed strategy. Intuitively, $\text{FixedGreedy}$ performs badly in the case of multiple queries because the placement of filters in the strategy does not take into account the participation of each filter, i.e., the number of queries in which the filter participates. In fact, our main adaptive strategy presented in Section 2.4 can be viewed as an extension of Theorem 2.3.3 to take participation into account.

Fixed strategies have a low execution overhead: they only incur the usual bookkeeping cost of $O(\lambda)$ for each data item (recall Section 2.2.1). To avoid redundant work, when a filter $F$ is chosen to be evaluated, we iterate through all the queries in which $F$ participates. If all these queries are resolved, evaluation of $F$ is skipped. The cost of these iterations is also of the same order as the bookkeeping cost (at most $O(\lambda)$ for each data item). Apart from this, the decision step in Line 5 of Figure 2.1 is very cheap, since filters are evaluated in a fixed order. However, in terms of the expected total filter evaluation cost (given by (2.2)), fixed strategies are often inferior to the more general adaptive strategies, as shown next.

2.3.2 Adaptive Strategies

Recall Definition 2.2.1 that defines general adaptive strategies. An adaptive strategy can be represented conveniently as a decision tree in which each node corresponds to a filter and has two children referred to as the Yes child and the No child respectively.
2.3. SHARED EXECUTION STRATEGIES

To build the decision tree for a given adaptive strategy $\mathcal{P}$, note that the first filter to be evaluated by $\mathcal{P}$ must be a fixed filter $F$ (since there are no previous filters on which $F$ can depend). Make $F$ the root of the decision tree. Add as the Yes child of $F$ the filter that $\mathcal{P}$ evaluates next if $F$ evaluates to true. If the evaluation of $F$ to true resolves all queries and there are no more filters to be evaluated, the Yes child of $F$ is a special leaf node marked \( \perp \). The No child of $F$ is constructed similarly based on what happens if $F$ evaluates to false. The subtrees of the Yes and No children are then built recursively. Note that multiple nodes of the decision tree may correspond to the same filter.

**Example 2.3.5.** We continue with our running example (Example 2.2.2). Recall the queries $Q_1 = \{F_1, F_2\}$ and $Q_2 = \{F_2, F_3\}$. Consider the adaptive strategy $\mathcal{P}$ that first evaluates $F_1$. If $F_1$ evaluates to true, $\mathcal{P}$ evaluates $F_2$, otherwise it evaluates $F_3$. The third remaining filter is then evaluated if needed. The decision tree corresponding to $\mathcal{P}$ is shown in Figure 2.2, with nodes named $N_1, \ldots, N_7$.

The execution of $\mathcal{P}$ for each data item can be viewed as starting at the root of the corresponding decision tree and traversing to one of the leaves based on filter results. We assume that no two nodes on the path from the root to a leaf can correspond to the same filter (since that would constitute redundant filter evaluation). Let $\Pr[N]$ denote the probability that a node $N$ is visited during execution of $\mathcal{P}$. Let $N_{yes}$ and $N_{no}$ denote the children of $N$. If the filter at $N$ is $F_i$, we have (by using $\Pr[F_i = \text{true}] = s_i$ and independence of filters):

\[
\begin{align*}
\Pr[N_{yes}] &= \Pr[N] \cdot s_i \\
\Pr[N_{no}] &= \Pr[N] \cdot (1 - s_i)
\end{align*}
\]

Thus, given that $\Pr[\text{root}] = 1$ (since the root is always visited), $\Pr[N]$ can be calculated for each node $N$. For Example 2.3.5, $\Pr[N]$ for each node $N$ is annotated in Figure 2.2.
As in the case of fixed strategies, to calculate the cost of an adaptive strategy $\mathcal{P}$, we must calculate the probability $e_i$ that $\mathcal{P}$ evaluates filter $F_i$ (recall (2.2)). If filter $F_i$ occurs at nodes $N^1_i, \ldots, N^k_i$ of the decision tree, then $e_i$ is given by

$$e_i = \sum_{j=1}^k \Pr[N^j_i]$$

(2.5)

Example 2.3.6. We continue with our running example (Example 2.2.2) and the adaptive strategy $\mathcal{P}$ shown in Figure 2.2. Since $F_1$ occurs at the root, $e_1 = 1$. $F_2$ occurs at nodes $N_2$ and $N_5$. From (2.5), $e_2 = \Pr[N_2] + \Pr[N_5] = s_1 + (1 - s_1)s_3$. Similarly, $e_3 = 1 - s_1 + s_1s_2$. From (2.2):

$$\text{cost}(\mathcal{P}) = c_1 + (s_1 + (1 - s_1)s_3) \cdot c_2 + (1 - s_1 + s_1s_2) \cdot c_3$$

We now motivate the need for designing adaptive strategies by showing that they are vastly superior in performance to any fixed strategy when the sharing parameter $\mu$ is large.

Theorem 2.3.7. There exist problem instances where the cost of any fixed strategy is $\Omega(\mu)$ times the cost of the optimal adaptive strategy.

Proof. We construct a problem instance as follows. Let $n = \sqrt{m}$. We have $n$ filters $F_1, F_2, \ldots, F_n$ with cost $c = 0$ and selectivity $s = 1/n$. There are another $n$ filters $H_1, \ldots, H_n$ of cost $c = 1$ and selectivity $s = 0$. There are $m$ queries $Q_1, Q_2, \ldots, Q_m$ which are divided into $n$ disjoint groups $G_1, G_2, \ldots, G_n$. Filters $F_i$ and $H_i$ are present in all queries in group $G_i$. In addition, we add each filter $H_i$ to one query from each of the remaining groups. This addition is done such that any query in group $G_j$ has at most one filter $H_i$ ($i \neq j$) mapped to it. Note that each query in group $G_i$ has at most 3 filters—$F_i, H_i$ and one $H_j$ for some $j \neq i$. Thus, $\kappa = 3$. Also, $\mu = \Theta(n)$. The construction is illustrated in Figure 2.3 for the case $m = 9$, and $n = 3$.

Without loss of generality, any strategy first evaluates all the filters $F_1, \ldots, F_n$ since these filters have zero cost. Since the selectivity of these filters is $1/n$, in expectation one group of queries remains unresolved at the end of this evaluation. The best adaptive strategy then evaluates the filters $H_i$ corresponding to the unresolved groups $G_i$, spending unit cost for each unresolved group. Since the expected number of unresolved groups is 1, the expected cost of the best adaptive strategy is $O(1)$.

Now consider any fixed strategy. It has to choose an ordering of $H_1, \ldots, H_n$ in advance, and at best can choose a random ordering since the instance is symmetric on the indices of
the filters. Fix the event that exactly one group of queries $G_{j}^*$ is unresolved after evaluating $F_1, \ldots, F_n$. This event happens with probability $1/e$. The filter $H_{j}^*$ appears at location $n/2$ in the ordering of $H_1, \ldots, H_n$ in expectation. All filters $H_i$ before $H_{j}^*$ in the ordering need to be evaluated since they are present in some query in the unresolved group $G_{j}^*$. Therefore, the expected cost of any fixed strategy is $\Omega(n) = \Omega(\mu)$.

The bound given by the above theorem is tight since we have already seen (in Theorem 2.3.4) that the fixed strategy $\text{FixedGreedy}$ is a $\mu$-approximation.

Adaptive strategies have higher execution overhead than fixed strategies since they incur the cost of deciding the next filter to be evaluated at each step. One way to avoid this cost is to store the entire decision tree corresponding to the strategy in memory. However, storing the entire decision tree is infeasible in general since its size can be exponential in the number of filters.

In the next section, we only focus on filter evaluation cost and consider the problem of finding the adaptive strategy that minimizes the expected total cost of filters evaluated for a given set of queries. Then in Section 2.5, we show how the execution overhead of an adaptive strategy can be reduced by appropriate precomputation.
CHAPTER 2. SHARING COMPUTATION AMONG CONTINUOUS QUERIES

2.4 Finding the Optimal Strategy

In this section, we consider the problem of finding the optimal strategy for a given set of queries (Definition 2.2.3). We first show the similarity of our problem to the well-known set cover problem (Section 2.4.1). Based on this similarity, we show the hardness of finding (or even approximating) the optimal strategy in the general case. We then give a lower bound on the cost of the optimal strategy (Section 2.4.2), followed by our general greedy adaptive strategy in Section 2.4.3. We analyze a variant of this greedy strategy in Section 2.4.4, and study the execution overhead of our adaptive strategy in Section 2.4.5.

2.4.1 Hardness and Set Cover

Definition 2.4.1 (Set Cover Problem [63]). Given a collection $S$ of $k$ sets $S_1, \ldots, S_k$, choose a minimum collection $C$ of sets from $S$ that covers the universal set, i.e., $\bigcup_{S_i \in C} S_i = \bigcup_{i=1}^{k} S_i$.

For our problem, let the universal set be the set of all queries $Q$, and let each filter $F_i$ be a set $S_i = \{ Q \in Q \mid F_i \in Q \}$. Thus, a filter covers all the queries that it can potentially resolve, and the aim is to pick the least-cost collection of filters that resolve (or cover) all the queries; hence the similarity to set cover. However, our problem departs from classical set cover in that the notion of a filter resolving (or covering) a query is probabilistic. Thus, when a filter $F_i$ is picked, it resolves the set of queries $S_i$ only with probability $1 - s_i$, i.e., when it evaluates to false. Otherwise, with probability $s_i$, i.e., when it evaluates to true, it resolves only those queries in $S_i$ in which it was the solitary filter. Thus our problem can be viewed as a new probabilistic version of set cover that, to the best of our knowledge, has not been considered before.

The following hardness result for our optimization problem follows easily from a reduction from set cover (recall that $m$ is the number of queries).

Theorem 2.4.2. No polynomial-time algorithm for finding an adaptive strategy can have an approximation ratio $o(\ln m)$, unless $P=NP$.

Proof. Consider the special case of our problem when all filters have cost $= 1$ and selectivity $= 0$. By the discussion in Section 2.4.1, finding the optimal strategy in this case is exactly equivalent to finding the optimal set cover. (Exact equivalence holds since all selectivities are 0, i.e., each filter definitely covers all its queries). Since set cover is NP-hard to approximate to any factor smaller than $\ln m$ [56], the result follows. \qed
2.4. FINDING THE OPTIMAL STRATEGY

2.4.2 Lower Bound on Cost

In this section, we prove a lower bound on the cost of the optimal strategy. This lower bound is used to bound the approximation ratio of our algorithm. It is also used in our experiments (Section 2.6) as a benchmark to compare our algorithm against.

For every query \( Q_i \in Q \), let \( r_i \) denote the probability that \( Q_i \) resolves to false. Then,

\[
r_i = 1 - \prod_{j | F_j \in Q_i} s_j
\]  

(2.6)

Consider the following linear program with variables \( e_1, \ldots, e_n \).

Minimize \( \sum_{j=1}^{n} c_j \cdot e_j \) subject to:

\[
\sum_{j \in Q_i} (1 - s_j) e_j \geq r_i
\]

\[
\forall \text{ queries } Q_i \in Q \quad \forall \text{ filters } F_j \in \mathcal{F}
\]

Theorem 2.4.3. The cost of the optimal strategy is lower bounded by the optimum value of the linear program in (2.7).

Proof. Consider any adaptive strategy \( \mathcal{P} \). Let \( e_j \) denote the probability that \( \mathcal{P} \) evaluates filter \( F_j \). By the union bound in probability theory, \( \Pr[Q_i \text{ resolves to false}] = r_i \leq \sum_{j | F_j \in Q_i} \Pr[F_j \text{ is evaluated } \land F_j = \text{ false}] \). Since \( F_j \) evaluating to false is independent of \( F_j \) being evaluated, we have:

\[
r_i \leq \sum_{j | F_j \in Q_i} (1 - s_j) e_j
\]

From (2.2), the cost of \( \mathcal{P} \) is \( \sum_{j=1}^{n} c_j \cdot e_j \). Thus, the optimum value of the linear program in (2.7) is a lower bound on the cost of any adaptive strategy. \( \square \)

The above linear program essentially approximates the probability of filter evaluations for any decision tree by using linear constraints that are derived by applying union bounds. This makes the lower bound efficiently computable. However, we have not encoded the constraint that the values of \( e_j \) must be obtainable from some decision tree by (2.5), which seems hard to do through a linear program. Thus, the above lower bound may be loose. However, for our problem, it turns out to be good enough for designing
Algorithm **Greedy**
1. if (exists unresolved query $Q$ with exactly 1 unevaluated filter)
2. pick the single filter in $Q$ to be evaluated next
3. else
4. $p_i ←$ number of unresolved queries $F_i$ is part of
5. $\text{pRank}(F_i) ← \frac{c_i}{p_i(1-s_i)}$
6. pick unevaluated filter with min pRank to be evaluated next

Figure 2.4: Greedy Adaptive Strategy

approximation schemes. (In our experiments, we found that this lower bound was only about a factor of 2 lower than the actual optimal cost.)

This general technique of lower bounding the value of an adaptive strategy using linear constraints is also used in the context of stochastic scheduling [47, 109] to develop non-adaptive strategies that are constant factor approximations to the best adaptive strategy. For our problem however, we use the linear program to design approximate adaptive strategies.

### 2.4.3 Greedy Adaptive Strategy

In this section, we describe our general greedy adaptive execution strategy and provide theoretical guarantees regarding its cost. Recall the similarity of our problem to set cover (Section 2.4.1). The following greedy algorithm is the best known polynomial-time algorithm for set cover: Start by picking the set that covers the maximum number of elements. Then at each stage pick the set that covers the maximum number of uncovered elements, and continue until all elements are covered.

Let us try using this algorithm to find an adaptive strategy for our problem, which is a probabilistic version of set cover. Our aim is to resolve all queries. Thus, at any stage, we should pick the filter that is expected to resolve the maximum number of unresolved queries per unit cost. This algorithm, **Greedy**, is shown in Figure 2.4 (written assuming it will be invoked repeatedly by Line 5 of algorithm **ExecuteStrategy** in Figure 2.1).

If there exists an unresolved query $Q$ that has only a single unevaluated filter $F$, then evaluation of $F$ is necessary to resolve $Q$. Hence we first evaluate any such filter $F$ (Lines 1-2 of Figure 2.4). Suppose filter $F_i$ occurs in $p_i$ unresolved queries. Then with probability
1 - s_i, F_i resolves p_i queries (when it evaluates to false), otherwise with probability s_i it does not resolve any queries (when it evaluates to true, since there are no queries with a single filter remaining). Thus the expected number of queries resolved by F_i is p_i(1 - s_i).

Analogous to Theorem 2.3.3, we define the pRank of F_i as the ratio of c_i to p_i(1 - s_i), and then pick the filter with the minimum pRank (Lines 4-6 of Figure 2.4). Thus, Greedy can also be seen as an extension of the technique in Theorem 2.3.3 to take participation of a filter into account, i.e., the number of queries in which a filter occurs.

If the filter selectivities are correlated, we need a simple modification to the definition of pRank. Let s_i denote the conditional selectivity of F_i given the results of all filters evaluated so far. Then pRank(F_i) ← c_i / (p_i(1 - s_i)), where p_i denotes the number of unresolved queries F_i participates in.

2.4.4 Analysis

The algorithm Greedy turns out to be hard to analyze since the number of adaptive decisions made is large. Instead, we have analyzed a natural variant of Greedy (called Greedy-Variant) that makes use of the bound in Section 2.4.2 more directly and hence is easier to analyze. Greedy-Variant is shown in Figure 2.5 (written assuming it will be invoked repeatedly by Line 5 of Algorithm ExecuteStrategy in Figure 2.1).

We refer to each invocation of Greedy-Variant as a phase. Greedy-Variant differs from Greedy in that in every phase, it returns a set of filters to be evaluated (Line 11) rather than a single filter at a time as returned by Greedy. Thus, Greedy-Variant evaluates chunks of filters non-adaptively before making the next adaptive decision (in contrast to Greedy that evaluates one filter before making the next adaptive decision).

Recall the definition of r_i from (2.6), the probability that query Q_i resolves to false. At any general stage of execution when some filters have already been evaluated, the current value of this probability is given by:

\[ r_i = 1 - \prod_{j \mid F_j \in Q_i \land F_j \text{ unevaluated}} s_j \]  

(2.8)

To describe and analyze Greedy-Variant we also define the following:

**Definition 2.4.4 (α-Satisfaction).** A set \( \mathcal{F}' \) of filters \( \alpha \)-satisfies a query \( Q \) if \( \sum_{j \mid F_j \in Q \cap \mathcal{F}'} (1 - s_j) \geq \alpha. \)
Algorithm **GreedyVariant**($\delta_1, \delta_3$)

1. $\bar{Q} \leftarrow$ set of unresolved queries
2. $Q_f \leftarrow \{Q_i \in Q \mid r_i \geq \delta_3\}$, $Q_t \leftarrow \bar{Q} - Q_f$
3. $F_l \leftarrow \{F \mid F \in Q \in Q_t \text{ and } F \text{ unevaluated}\}$, $F_h \leftarrow \phi$
4. for each $Q_i \in Q_f$, $\bar{r}_i \leftarrow r_i$
5. while ($Q_f$ not empty)
6. \(\text{pRank}(F_j) \leftarrow \frac{c_j}{\sum_{i \mid F_j \in Q_i} \min(r_i, 1 - s_j)}\)
7. $F_h \leftarrow F_h \cup \{\text{unevaluated filter } F_j \not\in F_h \text{ with min pRank}\}$
8. for each $Q_i \in Q_f$
9. \(\text{if } F_j^* \in Q_i \text{ then } \bar{r}_i \leftarrow \bar{r}_i - (1 - s_j^*)\)
10. \(\text{if } Q_i \text{ is } \delta_1\text{-satisfied by } F_h \text{ then } Q_f \leftarrow Q_f - \{Q_i\}\)
11. pick all filters in $F_l \cup F_h$ to be evaluated

Figure 2.5: Analyzed variant of Greedy

**Definition 2.4.5 (α-Cover).** An α-cover of a collection of queries $Q'$ is any set $F'$ of filters such that $F'$ α-satisfies $Q$ for every query $Q \in Q'$. The cost of the cover is $\sum_{j \mid F_j \in F'} c_j$. □

**Description of GreedyVariant**

**GreedyVariant** proceeds as follows. Fix three constants $0 < \delta_1 < \delta_2 < \delta_3 < 1$ such that $\delta_3 - \delta_2 \geq 0.1$ and $\delta_2 - \delta_1 \geq 0.1$. For concreteness, take them to be 0.25, 0.35 and 0.5 respectively. $\delta_2$ is used only in the analysis and not in the algorithm. First, we partition the set of unresolved queries $\bar{Q}$ into the set of queries $Q_f$ that probably resolve to false (indicated by a high value of $r_i \geq \delta_3$), and the remaining set of queries $Q_t$ that probably resolve to true (Line 2). Most of the unevaluated filters in the queries in $Q_t$ will need to be evaluated any way, so we pick all these filters to be evaluated (Line 3). Lines 6-7 are similar to greedily choosing the filter with the minimum pRank in Greedy (recall Figure 2.4), i.e., the filter that has the minimum ratio of cost to expected number of queries resolved. However, to calculate the expected number of queries resolved by filter $F_j$, we cannot simply say, as we did in Greedy, that filter $F_j$ resolves every unresolved query it is part of with probability $1 - s_j$: We are choosing a set of filters to evaluate rather than a single filter, and the probability that a query is unresolved may have been substantially reduced due to filters already chosen in the current phase. Thus, we maintain a current estimate $\bar{r}_i$ of the probability that $Q_i$ is unresolved and will eventually resolve to false. This estimate...
is updated whenever a filter is added to the set of filters chosen to be evaluated. Then, if \( F_j \in Q_i \), \( F_j \) resolves \( Q_i \) only with probability \( \min(\bar{r}_i, 1 - s_j) \) rather than with probability \( 1 - s_j \). Clearly, \( \bar{r}_i \) is initially \( r_i \) (Line 4) and reduces by \( 1 - s_{j^*} \) when \( F_{j^*} \in Q_i \) is chosen to be evaluated (Line 9). We continue choosing filters with the minimum pRank in this way until the probability that \( Q_i \) is unresolved has decreased sufficiently for every query \( Q_i \in Q_f \) (Line 10).

**Analysis of GreedyVariant**

Let OPT denote the expected total cost of filters evaluated by the optimal adaptive strategy. Intuitively, the proof rests on proving the following two main points:

1. In every phase, the cost of filters chosen to be evaluated is at most \( O(\log^2 m) \) times OPT.

2. After every phase, the number of unresolved queries decreases by at least a constant factor, thus the number of phases is logarithmically bounded.

Combining the above two, we shall show that GreedyVariant is an \( O(\log^2 m \log n) \) approximation. Let a **partial decision tree** be a decision tree where the leaf nodes may have unresolved queries.

**Lemma 2.4.6.** Let \( T \) be any partial decision tree. The optimal expected cost of extending \( T \) to a complete decision tree (where all queries are resolved at any leaf node) is \( \leq \) OPT.

**Proof.** This lemma follows from the intuitive fact that even if some filters have been evaluated and some queries resolved (as given by the partial decision tree \( T \)), we can still follow the optimal strategy to resolve all the queries, skipping the evaluation of filters that had already been evaluated in \( T \) and reusing the results of evaluation from \( T \). Clearly, the cost of doing so cannot be greater than the original optimal cost OPT, since the evaluation of some filters is skipped.

**Lemma 2.4.7.** The cost of the filters in \( F_i \) is at most \( \frac{1}{1 - \delta_3} \) OPT.

**Proof.** Any query \( Q \) in \( Q_t \) resolves to true with probability at least \( 1 - \delta_3 \). Thus the optimal strategy evaluates all filters in the queries in \( Q_t \) with probability at least \( 1 - \delta_3 \). Thus, OPT \( \geq (1 - \delta_3) \sum_{j \mid F_j \in Q_t} c_j = (1 - \delta_3) \sum_{j \mid F_j \in F_i} c_j \).
Lemma 2.4.8. There exists a \( \delta_2 \)-cover for \( Q_f \) with cost is at most \( \text{OPT} \cdot O(\log m) \).

Proof. Consider the fractional \( e_j \) obtained by solving the linear program in (2.7). As shown in Section 2.4.2, the optimal value of this linear program is a lower bound on \( \text{OPT} \). We first set \( e_j' \leftarrow \min(1, 30e_j \log m) \). Clearly, the cost of the resulting solution \( \sum_{j=1}^n c_j e_j' \leq 30 \log m \cdot \text{OPT} \). Let \( F_A \) be the set of filters \( F_j \) for which \( e_j' = 1 \), and \( F_B \) be the set of remaining filters, i.e., \( F_B = F - F_A \).

If for some query \( Q_i \in Q_f \), \( \sum_{j \in F_A \cap Q_i} (1 - s_j) \geq \delta_2 \), then \( Q_i \) is already \( \delta_2 \)-satisfied by \( F_A \) alone. Now consider the remaining queries \( Q_i \in Q_f \). For these queries, we have:

\[
\sum_{j \in F_j \in F_A \cap Q_i} (1 - s_j) < \delta_2
\]  

(2.9)

By the first constraint in the linear program in (2.7), and because \( r_i \geq \delta_3 \), and for all \( j \), \( e_j \leq 1 \), we have:

\[
\sum_{j \in F_j \in F_A \cap Q_i} (1 - s_j) + \sum_{j \in F_j \in F_B \cap Q_i} (1 - s_j)e_j \geq \delta_3
\]

Combining the above with (2.9), and because \( \delta_3 - \delta_2 \geq 0.1 \), we get \( \sum_{j \in F_j \in F_B \cap Q_i} (1 - s_j)e_j \geq 0.1 \). Finally, since for filters \( F_j \in F_B \) \( e_j' = 30e_j \log m \), we get:

\[
\sum_{j \in F_j \in F_B \cap Q_i} (1 - s_j)e_j' > 3 \log m
\]  

(2.10)

For filters in \( F_B \), we perform randomized rounding [124] of \( e_j' \), setting \( e_j' \) to 1 with probability equal to \( e_j' \) and to 0 otherwise. The expected cost of the resulting solution \( \sum_{j=1}^n c_j e_j' \) remains the same, i.e., \( \leq 30 \log m \cdot \text{OPT} \). By Chernoff bound [97] and using (2.10), for any \( Q_i \in Q_f \) that was not already \( \delta_2 \)-satisfied by \( F_A \), the probability that after rounding \( \sum_{j \in F_j \in F_B \cap Q_i} (1 - s_j)e_j' < 1 \) is \( O(1/m^2) \). Let \( F_C = F_A \cup \{ F_j \in F_B \mid e_j' \} \) was rounded to 1 \}. Thus all queries in \( Q_f \) are \( \delta_2 \)-satisfied by \( F_C \) with probability at least \( 1 - 1/m \). Since the cost of \( F_C \) is at most \( \text{OPT} \cdot O(\log m) \), it is the required \( \delta_2 \)-cover. \( \square \)

Lemma 2.4.9. At the end of the first phase, the cost of the filters in \( F_h \) is at most \( O(\log m) \) times the cost of the optimal \( \delta_2 \)-cover.

Proof. Let \( C \) be the cost of the optimal \( \delta_2 \)-cover. The sum \( U = \sum_{Q_i \in Q_f} \bar{r}_i \) is at most \( m \) initially. By the same argument as the proof for greedy set cover [124], it can be shown that the greedy step in Line 6 (Figure 2.5) ensures that every time filters of cost at most \( C \)
are added to $F_h$, $U$ reduces by at least a factor half. When $U$ falls below $\delta_3 - \delta_1$, the loop in lines 5-10 terminates since $F_h$ is now a $\delta_1$-cover for the queries in $Q_f$. This is because $U \leq \delta_3 - \delta_1$ implies that for each query $Q_i \in Q_f$, $\bar{r}_i \leq \delta_3 - \delta_1$, i.e., $\bar{r}_i$ has decreased by at least $\delta_1$ from its initial value, thereby implying that $F_h$ $\delta_1$-satisfies $Q_i$. Thus, $U$ needs to be reduced by a factor half at most $\log(\frac{m}{\delta_3 - \delta_1})$ times. Since each reduction adds filters of cost at most $C$ to $F_h$, the cost of $F_h$ at the end is at most $O(C \log m)$.

**Lemma 2.4.10.** The cost of filters evaluated in the first phase is at most $\text{OPT} \leq O(\log^2 m)$.

*Proof.* From Lemmas 2.4.8 and 2.4.9, the cost of filters in $F_h$ is at most $\text{OPT} \leq O(\log^2 m)$. Combining with Lemma 2.4.7, we get the result.

**Lemma 2.4.11.** The expected number of unresolved queries at the end of the first phase is at most $e^{-\delta_1 Q_f}$.

*Proof.* All the queries in $Q_t$ are definitely resolved after the first phase since all their un-evaluated filters ($F_l$) are chosen to be evaluated. Now consider the queries in $Q_f$. Since $F_h$ is a $\delta_1$-cover for $Q_f$, for every query $Q_i \in Q_f$, we have $\sum_j |F_j \cap F_h| (1 - s_j) \geq \delta_1$. Thus the probability that $Q_i$ is left unresolved after evaluating all filters in $F_h$ is at most $\prod_j |F_j \cap F_h| s_j \leq e^{-\sum_j |F_j \cap F_h| (1 - s_j)} \leq e^{-\delta_1}$. Thus the number of unresolved queries at the end of the first phase is at most $e^{-\delta_1 Q_f}$.

**Theorem 2.4.12.** GreedyVariant is a $O(\log^2 m \log n)$ approximation to the optimal adaptive solution.

*Proof.* By Lemma 2.4.11, the expected number of unresolved queries reduces by at least a constant factor. Since this reduction is independent of the queries unresolved at the beginning, Lemma 2.4.11 can be applied to any phase. Suppose $c_{\max}$ and $c_{\min}$ are the maximum and minimum filter costs. After $\log(n c_{\max}/c_{\min})$ phases, the expected number of unresolved queries is at most $O(\frac{c_{\min}}{n c_{\max}})$. Then, even if we apply all filters non-adaptively, we could apply at most $n$ filters of cost at most $c_{\max}$. The expected cost of this application is therefore at most $O(c_{\min})$, which can be ignored since OPT is also at least $c_{\min}$.

By Lemma 2.4.6, the expected cost of the optimal solution given a partial decision tree is at most OPT. Combining with Lemma 2.4.10, we obtain that for every phase, the expected cost is at most $O(\text{OPT} \cdot \log^2 m)$. Therefore, GreedyVariant is a $O(\log^2 m \cdot \log n)$ approximation to OPT. If $c_{\max}/c_{\min}$ is polynomially bounded in $n$, the approximation ratio is $O(\log^2 m \log n)$. 


To get an approximation ratio independent of the costs of the filters in the general case, we group the filters based on cost into powers of \( n^3 \) (assuming by scaling that \( c_{\text{min}} = 1 \)). There are at most \( n \) groups, since there are \( n \) filters. We then separately bound the contribution of each cost group to the cost of GreedyVariant, thereby obtaining a final approximation ratio of \( O(\log^3 m \log n) \).

Note that although the above analysis assumes independent filters, we have performed experiments for both independent and correlated filters. In our experiments, GreedyVariant always performs worse than Greedy (although only slightly so), which gives strong evidence that Greedy itself has similar theoretical guarantees. In practice, Greedy performs very well and produces the optimal solution on most instances. We now consider the execution overhead of Greedy at each invocation.

### 2.4.5 Execution Overhead of Greedy

**Theorem 2.4.13.** The per-tuple execution overhead of algorithm Greedy is \( O(\lambda \log n) \) for independent filters and \( (\lambda + n^2 t) \log n \) for correlated filters, where \( t \) is the time for computing a conditional selectivity value. Recall that \( \lambda \) is the total number of \((F_i, Q_j)\) pairs where \( F_i \) is present in \( Q_j \).

**Proof.** We maintain a priority queue of filters sorted by pRank. Each time a filter is evaluated, consider the queries that this filter participates in. If these queries are resolved, they are deleted. The total work involved in deleting resolved queries is \( O(\lambda) \). The pRanks of all filters that participate in these queries needs to be updated. For a given query, this operation happens once when the query is satisfied. Therefore, the total number of times the pRank of filters is updated is \( O(\lambda) \). Each of these updates involves manipulating the heap, which takes \( O(\log n) \) time.

If filter selectivities are correlated, each time a filter is evaluated, the pRanks of all other filters needs to be updated since their conditional selectivity changes. The number of such updates is \( O(n^2) \). The total amount of work is now \( O((\lambda + n^2 t) \log n) \). We assume that computing the new selectivity takes \( t \) time; in practice, for simple types of correlation, this computation can be implemented in \( O(1) \) time using table lookup.

Note that the execution overhead of Greedy is only \( O(\log n) \) times worse than the fixed bookkeeping overhead. This however can be significant, especially for cheap filters. We now show how the execution overhead of Greedy (or any adaptive strategy in general) can be reduced by precomputation.
2.5 Reducing Execution Overhead

One way to reduce the execution overhead of an adaptive strategy \( \mathcal{P} \), given arbitrary amounts of memory, is to precompute and store the entire decision tree corresponding to \( \mathcal{P} \). Deciding the next filter to be evaluated is then a simple lookup into the stored decision tree. However, in general the size of the decision tree may be exponential in the number of filters, making it infeasible to store the entire decision tree unless the total number of filters \( |\mathcal{F}| \) is small. In practice, we may have sufficient memory to precompute and store only some \( M \) nodes of the decision tree, with the rest of the filter selections occurring dynamically. Our goal is to decide which \( M \) nodes from the decision tree to precompute and store, such that the expected execution overhead incurred is minimized.

Consider the execution overhead incurred when \( \mathcal{P} \) is executed with \( M \) stored nodes. Recall from Section 2.3.2 that the execution of an adaptive strategy for each data item can be viewed as starting at the root of the decision tree and traversing down to some leaf node. For any node \( N_i \), let \( o[N_i] \) denote the overhead incurred in computing node \( N_i \), and let \( \Pr[N_i] \) be the probability of visiting node \( N_i \) (from (2.4)). If \( N_i \) is stored, its contribution to the execution overhead is 0 since only a simple lookup is needed whenever \( N_i \) is visited. If \( N_i \) is not stored, the expected contribution of \( N_i \) to the overall execution overhead is given by:

\[
O[N_i] = \Pr[N_i] \cdot o[N_i]
\]

Clearly, to minimize the expected overhead given space for storing only \( M \) nodes, we must store those nodes that have the \( M \) highest values for \( O[N_i] \).

In our experiments, we assume \( o[N_i] = c \) for every node \( N_i \) where \( c \) is some constant. However, our algorithm only requires that \( o[N_i] \) decreases as we go down the decision tree, i.e., if node \( N_i \) is an ancestor of node \( N_j \), then \( o[N_i] \geq o[N_j] \). This assumption is reasonable because the overhead of deciding the next filter often depends on the number of unevaluated filters (as in Greedy) or the number of unresolved queries, both of which decrease as we go down the decision tree. Note that the probability of visiting nodes also decreases as we go down the tree (from (2.4)).

**Theorem 2.5.1.** Algorithm Precompute (Figure 2.6) decides the \( M \) nodes to store such that the execution overhead is minimized.

**Proof.** Since \( o[N_i] \) and \( \Pr[N_i] \) both decrease as we go down the decision tree, \( O[N_i] \) also
Algorithm *Precompute* \((M, \mathcal{P})\)

\(M\): number of decision-tree nodes that can be stored

\(\mathcal{P}\): Adaptive strategy \(\mathcal{P}\)

1. candidates ← \{root of decision tree corresponding to \(\mathcal{P}\}\}
2. while (number of stored nodes < \(M\))
3. store node \(N_i \in\) candidates with maximum \(O[N_i]\) (from (2.11))
4. candidates ← candidates − \{\(N_i\)\} ∪ children of \(N_i\)

Figure 2.6: Algorithm for Precomputing an Adaptive strategy

decreases on going down the decision tree. Hence any node in the subtree of node \(N_i\) need not be considered for storing unless \(N_i\) has already been stored. Thus, algorithm \(Precompute\) starts by considering only the root for storing, and then every time a node \(N_i\) is chosen for storing, its children are added to the set of nodes being considered for storing. This process continue until \(M\) nodes have been chosen, and requires only \(O(M)\) node computations.

2.6 Experiments

We now present an experimental evaluation of our techniques. We carried out two broad categories of experiments—one measuring the expected total cost of filters evaluated by a strategy, and the other measuring the execution overhead incurred by a strategy. For our cost experiments, we compared our shared execution strategy *Greedy* (Figure 2.4) against the following two naïve execution strategies\(^2\):

1. *Naïve*: This strategy models what a system that optimizes queries one at a time would do when multiple queries of the form (2.1) are registered. Each query executes in isolation from other queries and evaluates its filters in rank order for every data item (Theorem 2.3.3). Filter evaluations are not shared across queries.

2. *Shared*: This strategy models a first cut towards sharing. Each query evaluates its filters in rank order as in *Naïve*. If a query \(Q\) needs to evaluate a filter \(F\) that has already been evaluated by some other query, \(Q\) reuses the previous result.

\(^2\)We do not report cost numbers for *GreedyVariant* as they were very close to those for *Greedy*.
query evaluates one filter at a time with round robin execution among the queries. Additionally, when a filter $F$ evaluates to false, all queries containing $F$ are marked as resolved and do not execute any further for that data item.

For our execution overhead experiments, we included another execution strategy called 
\textit{Cache} in our comparison. 
\textit{Cache} is \textit{Greedy} augmented with a fixed-size cache to precompute and store some nodes of the corresponding decision tree according to Algorithm \textit{Precompute} in Figure 2.6.

To capture a wide range of scenarios, we experimented with synthetic as well as real problem instances. The main findings from our experiments are:

1. For the problem instances we generated, the expected total cost of filters evaluated by \textit{Greedy} is about twice the optimal cost, whereas for \textit{Shared} it can be up to 8 times the optimal cost, and for \textit{Na"ive} up to 12 times the optimal cost.

2. As expected, \textit{Greedy} has a higher execution overhead than \textit{Shared}. However, the execution overhead can be substantially reduced (comparable to \textit{Shared}) by using \textit{Cache}.

We first describe how we generated problem instances for our experiments (Section 2.6.1). We report on our cost experiments in Section 2.6.2 and our overhead experiments in Section 2.6.3.

2.6.1 Problem Instance Generation

We built a \textit{problem instance generator} with the following input parameters:

\begin{tabular}{|c|c|}
\hline
$n, m$ & Number of filters and queries respectively \\
\hline
$c, s, p$ & Range of filter costs, selectivities, and participation respectively \\
\hline
\end{tabular}

Table 2.2: Input Parameters to Problem Instance Generator

The generator first creates a set $\mathcal{F}$ of $n$ filters and a set $\mathcal{Q}$ of $m$ empty queries. For each filter in $\mathcal{F}$, the cost, selectivity, and participation (the number of queries it is part of) are chosen uniformly at random from the ranges $\tilde{c}$ and $\tilde{s}$ and $\tilde{p}$ respectively. Let $\bar{p}$ be the midpoint of the range $\tilde{p}$. Then on average each filter occurs in $\bar{p}$ queries.

Each query $Q_i \in \mathcal{Q}$ is assigned a weight $w_i$ chosen uniformly at random from $[0, 1]$. Each filter $F_j \in \mathcal{F}$ (having participation $p_j$) is then added to a randomly chosen set of $p_j$
queries from \( Q \) with query \( Q_i \) being chosen with probability proportional to \( w_i \). Thus, at the end of the generation process, the expected number of filters in a query is proportional to its weight. Assigning weights to queries avoids always generating problem instances in which all queries have roughly equal number of filters. The average number of filters per query (across all queries) is \( np/m \).

For our synthetic problem scenarios, filter evaluation is simulated by randomly deciding whether a filter evaluates to true or false, with \( \Pr[F_j \text{ evaluates to true}] = s_j \), independently for every filter (except for Section 2.6.2.5 where we experiment with correlated filters).

### 2.6.2 Cost Experiments

In Sections 2.6.2.1 to 2.6.2.4, we experiment with synthetic problem instances, varying the input parameters to our problem instance generator, and studying their effect on the expected total cost of filters evaluated by the execution strategies \textit{Greedy}, \textit{Shared}, and \textit{Naïve}. Then in Section 2.6.2.6, we report our results on a real problem instance.

For each generated problem instance \( I \), and for each strategy \( P \) being compared, we first measured the average cost of \( P \) on \( I \) by simulating the execution of \( P \) on 1000 input tuples. We found that 1000 tuples were enough to measure the average cost accurately since even on increasing the number of tuples, the measured average cost did not undergo any considerable change (typically within 2%). We then computed a lower bound on the cost of the optimal strategy on \( I \) by using the linear program in Section 2.4.2. Finally, we computed the ratio of the average cost of \( P \) on \( I \) to this lower bound (referred to as the \textit{approximation ratio} of \( P \) on \( I \)). Note that in reality this ratio is only an upper bound on the actual approximation ratio and not the exact approximation ratio, since it is obtained by using a lower bound on the optimal cost and not the exact optimal cost (which seems hard to find without an exhaustive search).

To compare various strategies for a particular setting of parameters, we randomly generated 100 problem instances with the same parameter settings using our problem instance generator (Section 2.6.1). For strategy \( P \), we then use the worst (or maximum) approximation ratio of \( P \) on any of these 100 instances as a metric for comparing \( P \) with other strategies. We use the maximum and not the average approximation ratio as a metric since for an execution strategy to be good, it is not enough if it performs well on average across all problem instances, but it should perform well on any given problem instance. Note that
our use of the maximum approximation ratio as a metric for comparison does not imply that we are focussing on the worst-case performance of an execution strategy as opposed to its average-case performance. For any given problem instance, we still consider the average cost of filters evaluated by a strategy, and not the worst-case cost.

We now present the results of our cost experiments. Unless varied for a specific experiment, the parameters for the problem instance generator were fixed at \( n = 150, m = 100, \tilde{c} = [8, 10], \tilde{s} = [0, 0.2], \) and \( \tilde{p} = [1, 100]. \)

### 2.6.2.1 Effect of Filter Costs

Figure 2.7 shows the worst approximation ratio of the various execution strategies as the range of filter costs \( \tilde{c} \) is varied from \([0, 10]\) to \([10, 10]\). *Greedy* consistently has an approximation ratio of about 2, independent of \( \tilde{c} \), whereas the approximation ratio of both *Naïve* and *Shared* generally worsens as the range of filter costs narrows: When the range of filter costs is wider, there are some cheap filters that might resolve queries at a low cost even for the naïve algorithms. However, as all filters become expensive, the naïve algorithms incur a high cost that is reflected in a higher approximation ratio.
2.6.2.2 Effect of Filter Selectivities

Figure 2.7 shows the worst approximation ratio of the various execution strategies as the range of filter selectivities $\tilde{s}$ is varied from $[0, 0.01]$ to $[0, 1]$. Both Naïve and Shared perform very badly when all filters have extremely low selectivity. However, as the range $\tilde{c}$ widens to $[0, 1]$, their performance improves: When all filters have low selectivity, significant gains can be achieved by choosing the right set of filters to evaluate first (since they can resolve most of the queries). As we include filters with higher selectivity, queries cannot be resolved early and the choice of the order of evaluation does not matter as much (e.g., in the extreme case when all filters have selectivity 1, any strategy must evaluate all filters). Our strategy Greedy is able to perform well (with an approximation ratio of about 2) independent of the range $\tilde{s}$.

2.6.2.3 Effect of Filter Participation

Figure 2.9 shows the worst approximation ratio of the various execution strategies as the range of filter participation $\tilde{p}$ is varied from $[1, 100]$ to $[80, 100]$. As in previous experiments, Greedy consistently has an approximation ratio of about 2 independent of $\tilde{p}$. However, Naïve and Shared perform very badly when $\tilde{p}$ is wide, and their performance improves only when $\tilde{p}$ becomes narrow: When $\tilde{p}$ is narrow, all filters have similar participation, and
the order of filters according to pRank as used by Greedy (Line 5 of Figure 2.4) is not too different from their order according to rank as used by Naïve and Shared (Theorem 2.3.3).

2.6.2.4 Effect of Number of Filters

In this experiment, our goal was to study the effect of the ratio of number of filters to number of queries \(n/m\) on the performance of various execution strategies. For this purpose, we varied \(n\) from 20 to 220 while \(m\) was held constant at 100. Figure 2.10 shows the worst approximation ratio of the various execution strategies. Greedy again has a consistent approximation ratio of about 2. However, the performance of both Naïve and Shared first worsens, and then improves as the number of filters is increased: When the number of filters is low, there is less room for mistakes in choosing which filters to evaluate first, and when the number of filters is high, there is a higher probability of having relatively cheap filters or low selectivity filters that can resolve queries early at a low cost. Thus, an intermediate number of filters is the worst case for Naïve and Shared.

2.6.2.5 Correlated Filters

In this final cost experiment on synthetic problem instances, our goal was to determine how our approach performs with correlated filters. To capture correlations among filters,
we use the relatively simple model introduced in [22]. In this model, the \( n \) filters are divided into \( \lceil n/k \rceil \) groups containing \( k \) filters each, where \( k \) is called the correlation index. If two filters belong to different groups, they are independent, otherwise they are positively correlated, agreeing on 80% of input tuples. Thus \( k = 1 \) implies independence, while \( k = n \) implies all filters are correlated with each other. On fixing \( k \) and an unconditional selectivity for each group, the conditional selectivities of the filters is implied by the 80% rule.

Figure 2.11 shows the worst approximation ratio of the various execution strategies as the correlation index is varied from 1 to 8. We also proportionally varied the total number of filters from 150 to 1200 so that the number of independent filter groups remained constant. Note that the versions of Shared and Naïve that we used for this experiment used conditional rather than unconditional selectivities to order filters (see footnote 2 in Section 2.3.1). The performance of all three strategies can be seen to be essentially independent of the correlation index: By using conditional selectivities instead of unconditional selectivities, all the three strategies are guarded equally against correlation, and since the number of independent filter groups remains constant, the performance does not change significantly.
2.6. EXPERIMENTS

2.6.2.6 Real Problem Instance

In this experiment, we applied filters on a stream of images and constructed queries using these filters. The filters we used were relatively simple but still expensive: each filter determined whether the amount of match in the image for a given color was more than a threshold or not (e.g., a large amount of green can be an indicator of vegetation). In reality, such filters may be more complicated [100] (e.g., they may perform texture matching to detect vegetation).

We used a collection of about 580 images of 800 × 600 resolution. We set up four filters $F_1, \ldots, F_4$ looking for different colors. Each filter had roughly the same cost. The selectivity of $F_1, \ldots, F_3$ on our image collection was about 1 in 20 while that of $F_4$ was about 3 in 20. We constructed three queries $\{F_1, F_4\}$, $\{F_2, F_4\}$, and $\{F_3, F_4\}$, and executed them over our collection. The filter processing time for various execution strategies was as follows (Table 2.3):

<table>
<thead>
<tr>
<th></th>
<th>Greedy</th>
<th>Shared</th>
<th>Naïve</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (msec)</td>
<td>3477.79</td>
<td>7070.48</td>
<td>7283.88</td>
</tr>
<tr>
<td>Average per image (msec)</td>
<td>6.01</td>
<td>12.21</td>
<td>12.58</td>
</tr>
</tbody>
</table>

Table 2.3: Performance over a Stream of Images
Even for this extremely simple scenario, Greedy yields about a factor of two improvement over Shared or Na"ive: Both Shared and Na"ive evaluate $F_1, \ldots, F_3$ before $F_4$ since $F_1, \ldots, F_3$ have lower rank, but Greedy recognizes that $F_4$ has a higher participation and evaluates it first. Due to the small number of filters and queries, the execution overhead in this experiment was practically zero.

### 2.6.3 Execution Overhead Experiments

In our execution overhead experiments, we varied the parameters to our problem instance generator and studied their effect on the average per-tuple execution overhead incurred by the strategies Greedy, Cache, and Shared. We also studied the effect of varying the number of nodes stored by Cache. We did not include Na"ive in our overhead comparisons since Na"ive requires no bookkeeping and hence incurs negligible execution overhead.

The average per-tuple execution overhead for a strategy $P$ was measured by simulating the execution of $P$ on 1000 input tuples. Similar to our cost experiments, for a particular setting of parameters, we randomly generated 100 problem instances and then used the maximum average overhead incurred by $P$ on any of these instances as a metric for comparing $P$ against other strategies.

Unless varied for a specific experiment, the parameters were fixed at $n = 300$, $m = 200$, $\bar{c} = [0, 10]$, $\bar{s} = [0, 1]$, and $\bar{p} = [0, 200]$. The number of nodes stored by Cache was fixed at 1000. The execution overhead incurred by a strategy is clearly independent of filter costs, hence we omit the experiments where $\bar{c}$ is varied. We also omit the experiments where $\bar{s}$ is varied, as they do not reveal any interesting trends except that as we include filters with higher selectivity, the execution overhead of all the three strategies uniformly increases (since more filters need to be evaluated to resolve queries).

#### 2.6.3.1 Effect of Filter Participation

Figure 2.12 shows the execution overhead of the various strategies as the range of filter participation $\bar{p}$ is varied from $[0, 200]$ to $[160, 200]$. Shared maintains a very low execution overhead even for high values of participation. However, the overhead of Greedy increases with increasing participation because then every query has a higher number of filters and every time a query is resolved, we have to iterate through all the filters to maintain their
pRanks. But the execution overhead of Cache remains low even with increasing participation since the above computation by Greedy is avoided as long as the next filter to be evaluated is given by a node that is cached.

We omit the results of the experiment in which \( n \) was varied, since the results were very similar to Figure 2.12: As the number of filters increases, the net effect on overhead is only due to an increase in the average number of filters per query—the same effect as that obtained by increasing the participation of every filter.

### 2.6.3.2 Effect of Number of Nodes Stored

Figure 2.13 shows how the execution overhead of Cache varies as the number of nodes of the decision tree stored by Cache is varied from 20 to 800. The constant execution overheads of Greedy and Shared for this experiment are also shown in Figure 2.13 for comparison. Much of the benefit of precomputation can be realized even with a small number of nodes being stored (as low as 400): There are only a few nodes in the decision tree that have a high probability of being visited, and precomputing them is enough to save much of the overhead incurred.
2.7 Related Work

There has been work pertaining to shared query execution in both the data streams context as well as the classical DBMS context. For data streams, most of the work on shared query execution is for publish-subscribe systems, e.g., [38, 116, 129]. In publish-subscribe systems, the problem setting is essentially the same as ours: Each user specifies (or subscribes to) the data of interest by specifying conditions on the incoming data stream, and the goal is to dispatch each incoming data item to all matching subscriptions. Publish-subscribe systems generally focus on scenarios where the number of subscriptions is very large (of the order of millions), and they use special disk-based indexing mechanisms to efficiently locate the matching subscriptions for a given data item. Unlike our work, their focus is not on deciding the order of filter evaluation, since filters are considered cheap. Another important distinction is that publish-subscribe systems often exploit the internal structure of filters to achieve sharing [74]. Such techniques to exploit sharing do not extend to the “black-box” user-defined filters we consider, or to other general filters such as those implemented by a table lookup or a semijoin. In the continuous query context, exploiting sharing for continuous sliding-window aggregates, but not for expensive filters, is considered in [14]. Expensive filters are considered in [22], but only for a single query.

In conventional relational DBMSs, shared query execution has been considered under
multi-query optimization [44]. This work focuses mostly on exploiting common join subexpressions in queries, and does not consider expensive filters. The techniques developed in this chapter can also be applied in the classical relational setting for optimizing multiple queries with expensive filters. Optimization of a single query with expensive filters has been considered in [36, 76]. In the context of active databases, the Ariel system [75] has been designed for efficiently evaluating a collection of rules or triggers using discrimination networks. However, just as in publish-subscribe systems, sharing is achieved by exploiting the internal structure of the trigger conditions. Furthermore, optimization in Ariel is mainly through randomization and hill-climbing with no provable guarantees of optimality.

Our notion of an adaptive execution strategy is not related to adaptive plans in Eddies [15], or even Eddies with content-based routing [24]. In Eddies, different plans may be chosen for different tuples in order to adapt to changes in operator costs and selectivities over time. In our case, we assume that operator costs and selectivities do not change over time. Rather, different plans may be chosen for different tuples since the next operator (filter) to be evaluated is based on the results of the filters evaluated so far. There has been work on sharing in the Eddies context [90], but this work does not provide provable performance guarantees.

We note that our problem is similar in spirit to [55], where adaptive querying of information sources is considered, each of which answers a subset of queries with some probability, time delay and cost. The significant difference with our work is that they assume each information source (predicate in our case) satisfies each query with some probability which is independent of its satisfying another query; in our setting, the filter satisfies all queries it belongs to with the same probability in a perfectly correlated fashion. This makes our problem harder to approximate and we need different solution strategies and analysis techniques.

2.8 Conclusion

In this chapter, we considered the problem of sharing computation among a collection of queries where each query is a conjunction of possibly expensive filters, and filters may be shared across queries. We explored the space of shared execution strategies and showed that for multiple queries, a fixed execution strategy (of the type commonly used for a single
query) may be suboptimal in terms of the expected total cost of filters evaluated. Instead, the optimal strategy for multiple queries is adaptive, i.e., the next filter to be evaluated is decided based on the results of the filters evaluated so far. We proved the hardness of approximating the optimal adaptive strategy to any factor smaller than logarithmic in the number of queries. We gave a simple greedy adaptive execution strategy that approximates the optimal strategy to within a factor polylogarithmic in the number of queries and filters. We also gave a simple method to reduce the execution overhead of any adaptive strategy. Finally, we gave a thorough experimental evaluation where we demonstrated that our greedy adaptive strategy pays a significantly lower cost of filter evaluations as compared to other naïve approaches towards exploiting sharing. Although our greedy strategy has a higher execution overhead than the more naïve approaches, its overhead can be drastically reduced by using precomputation.
Chapter 3

Memory-Limited Execution of Windowed Stream Joins

In the previous chapter, we gave techniques to share computation among continuous queries, in order to reduce consumed resources. Sharing of memory among continuous queries has been addressed in other work [13]. However, in spite of sharing, when stream rates are high and many continuous queries are registered, the resources available to a DSMS may not be sufficient to answer all registered queries exactly, without falling farther and farther behind. In such cases, for some applications it is acceptable for a DSMS to provide approximate answers to the extent possible given the available resources. In this chapter, we address the problem of providing approximate answers to a class of memory-intensive queries when the available memory is not sufficient for providing exact answers.

3.1 Introduction

Continuous queries often require maintaining a significant amount of state [10]. For example, a continuous query for the sum of the elements seen in the last one minute on a stream requires storing of all the elements seen in the last one minute. In the presence of a workload of numerous continuous queries over high-volume data streams, the available memory may not be sufficient for storing the aggregate state required by all queries [45, 81]. There are two basic solutions to this problem: provide exact results by using disk, with the risk of failing to keep up with the input rate [45, 123], or provide approximate instead of exact query results, using memory exclusively to ensure high performance [18, 45, 53]. Since
most data stream applications require immediate online answers, we focus on the latter alternative. In this chapter, we specifically address the problem of providing approximate answers to sliding-window join queries \cite{18}, when the available memory is insufficient to retain the entire state required by the join.

Consider a continuous sliding-window join between two streams $S_1$ and $S_2$, denoted as $S_1[W_1] \bowtie_{\theta} S_2[W_2]$. Windows $W_1$ and $W_2$ consist of the most recent tuples on their respective streams, and may be tuple-based (e.g., the last 1000 tuples), or time-based (e.g., tuples arriving in the last 10 minutes). The output of the join contains every pair of tuples from streams $S_1$ and $S_2$ that satisfy the join predicate $\theta$ and are simultaneously present in their respective windows. In general, to perform the join accurately, the entire contents of both windows must be maintained at all times. If we have many such joins with large windows over high-volume data streams, memory may be insufficient for maintaining all windows in their entirety. Even after sharing state, we must store, for every stream, the largest window required by any sliding-window-join query involving that stream. If memory is insufficient for accurate results, but the data stream application can tolerate an approximate results, there are two interesting types of approximation:

1. **“Max-Subset” Results**: If the application benefits from having a maximum subset of the result, we can selectively drop tuples (referred to as load shedding \cite{45,118}) with the objective of maximizing the size of the join result produced.

2. **Sampled Results**: A random sample of the join result may often be preferable to a larger sized but arbitrary subset of the result. For example, if the join result is being aggregated, the sample can be used to provide a consistent and unbiased estimate of the true aggregate.

Our first contribution is to show formally that if a sliding-window join over arbitrary streams is to be executed without enough memory for retaining the entire windows, neither of the above types of approximations can be carried out effectively: For the max-subset problem, any online algorithm can return an arbitrarily small subset as compared to the optimal (offline) algorithm \cite{45}, and for the sampling problem, no algorithm can guarantee a nonzero uniform random sample of the join result.

However, in the presence of additional information about stream arrival, it is possible to do better. For example, if we know that each join-attribute value has some roughly fixed frequency of occurrence on each stream (referred to as the frequency-based model), a
maximum subset of the join result can be effectively obtained. In [45,81], these frequencies of occurrence are used to decide which tuples to drop and which to retain, in order to maximize the accuracy of the result produced. However, there are many applications for which this frequency-based model is inappropriate. (One obvious case is a foreign-key join, where on one stream each value occurs at most once.) For these applications, we define an age-based model that is often appropriate and enables much more accurate query results. In the age-based model, the expected join multiplicity of a tuple depends on the time since its arrival rather than on its join-attribute value. Examples will be given in Section 3.2.2.

Given the two types of approximation and the two models, we have the problem space shown in Table 3.1. The max-subset problem has been addressed in previous work [45], but only for the frequency-based model. To the best of our knowledge, the sampling problem, i.e., the problem of extracting a random sample of the join result with limited memory, has not been addressed in previous work. Our contribution is to address the max-subset problem for the age-based model, and the sampling problem for both models.

Our discussion so far assumes a single two-way sliding-window join. In reality, we expect to be executing many queries simultaneously in the system. For windows on the same stream, sharing can be exploited. For other windows, there is an added dimension to all of the above problems: memory allocation among multiple joins. The total available memory should be allocated to the different joins such that a combined approximation measure is optimized. We provide an optimal memory allocation scheme that minimizes the maximum approximation error in any join. Our technique also extends to the weighted case, i.e., when different joins have different relative importance.

### Chapter Organization

The rest of the chapter is organized as follows. In Section 3.2, we introduce a novel age-based model for stream arrival that captures many applications not captured by the frequency-based model assumed in previous work. Then, for a single two-way join with a
fixed memory constraint, we provide novel algorithms for the max-subset problem under the age-based model (Section 3.3), and the sampling problem under both the frequency and age-based models (Section 3.4). Then, in Section 3.5, we move on to multiple two-way joins with an overall memory constraint, and give an algorithm to allocate memory among the various joins so as to optimize a combined measure of approximation. In Section 3.6, we provide a thorough experimental evaluation showing the effectiveness of our techniques. Finally, we survey related work in Section 3.7, and conclude in Section 3.8.

3.2 Preliminaries and Models

We briefly describe windows on streams and processing of two-way sliding-window joins over streams. Assume any discrete time domain. For a stream $S_i$, $i = 1, 2$, a variable number of tuples may arrive in each unit of time. However, we assume that over time, tuples on stream $S_i$ arrive at a constant average rate of $r_i$ tuples per unit time. $S_i[W_i]$ denotes a window on stream $S_i$. We consider time-based windows, where $W_i$ denotes the length of the window in time units. At time $t$, a tuple $s$ belongs to $S_i[W_i]$ if $s$ has arrived on $S_i$ in the time interval $[t - W_i, t]$. At time $t$, we say $s$ is of age $k$ if it arrived at time $t - k$. We consider time-based windows for generality; tuple-based windows can also be captured by assuming that a single tuple arrives every time unit.

The basic query we consider (shown in Figure 3.1) is a sliding-window equijoin between two streams $S_1$ and $S_2$ over a common attribute $A$, denoted $S_1[W_1] owtie_A S_2[W_2]$. The output of the join consists of all pairs of tuples $s_1 \in S_1$, $s_2 \in S_2$, such that $s_1.A = s_2.A$ and at some time $t$, both $s_1 \in S_1[W_1]$ and $s_2 \in S_2[W_2]$. Conceptually, a sliding-window
3.2. PRELIMINARIES AND MODELS

1. When a new tuple \( s \) arrives on \( S_1 \)
2. Update \( S_2[W_2] \) by discarding expired tuples
3. Emit \( s \bowtie_A S_2[W_2] \)
4. Add \( s \) to \( S_1[W_1] \)

Figure 3.2: Sliding-window join execution

The join is executed as shown in Figure 3.2, which details the steps to be followed for a newly arriving tuple on \( S_1 \). A symmetric procedure is followed for a newly arriving tuple on \( S_2 \).

If memory is limited, we need to modify the algorithm in Figure 3.2 in two ways. First, in Line 2, we update \( S_1[W_1] \) in addition to \( S_2[W_2] \) to free up memory occupied by expired tuples. More importantly, in Line 4, memory may be insufficient to add \( s \) to \( S_1[W_1] \). In this case, we need to decide whether \( s \) is to be discarded or admitted into \( S_1[W_1] \), and if it is to be admitted, which of the existing tuples is to be discarded. Discarding tuples earlier than their expiry is referred to as load shedding, and the algorithm that decides which tuple to discard is called a load-shedding strategy [20, 45, 118].

We also consider queries with a windowed aggregation operator on top of the streamed join result. Other work [46] has focused on approximate windowed aggregation in memory-limited environments. We do not consider this aspect of memory usage in our calculations, however analyzing the tradeoff between memory allocation to joins and aggregation is an interesting subject of future work.

We classify every join-result tuple as either an \( S_1 \)-probe join tuple or an \( S_2 \)-probe join tuple. When a new tuple \( s \) arrives on \( S_1 \) and joins with a tuple \( s' \in S_2[W_2] \) (line 3 of Figure 3.2), \( s \) and \( s' \) are said to produce an \( S_2 \)-probe join tuple. \( S_1 \)-probe join tuples are defined symmetrically. A tuple \( s \in S_1 \) may first produce \( S_2 \)-probe join tuples when it arrives. Then, before it expires from \( S_1[W_1] \), it may produce \( S_1 \)-probe join tuples with newly arriving tuples on \( S_2 \). We use \( n_i(s) \), \( i = 1, 2 \), to denote the number of \( S_i \)-probe join tuples produced by a tuple \( s \in S_i \) before it expires from \( S_i[W_i] \).

3.2.1 Frequency-Based Stream Model

Continue to consider the sliding-window join \( S_1[W_1] \bowtie_A S_2[W_2] \). Let \( D \) denote the domain of join attribute \( A \). The frequency-based model that has been assumed in previous work [45, 81] is defined as follows:

\[ \text{Join} \]
Definition 3.2.1 (Frequency-Based Model). For each value $v \in D$, a fixed fraction $f_1(v)$ of the tuples arriving on $S_1$, and a fixed fraction $f_2(v)$ of the tuples arriving on $S_2$, have value $v$ in attribute $A$.

Assuming an average rate $r_2$ of arrivals per unit time on $S_2$, the expected number of $S_1$-probe join tuples that a tuple $s \in S_1$ produces is given by:

$$E[n_1(s)] = r_2 \cdot W_1 \cdot f_2(s.A)$$ \hspace{1cm} (3.1)

Tuple $s$ remains in $S_1[W_1]$ for $W_1$ time units, during which $r_2 \cdot W_1$ tuples arrive on $S_2$, out of which a fraction $f_2(s.A)$ join with $s$.

As an example for the frequency-based model, suppose we are monitoring a system with a fixed number of components. We have a stream of actions and a stream of errors on all components, and we want to perform a sliding-window join on component-id to look for possible correlations between actions and errors. Some components may be more heavily used than others, and some may be more error-prone than others, but each component-id may have a roughly fixed frequency of occurrence on each stream.

### 3.2.2 Age-Based Stream Model

The frequency-based model is not appropriate for all applications. As a simple example, consider online auction monitoring [110] with the following streams:

- $S_1$: OpenAuction(auction-id,seller-id)
- $S_2$: Bid(auction-id,bid-amount)

When a seller starts an auction, a tuple arrives on $S_1$. When a bid is placed on an auction, a tuple arrives on $S_2$. Suppose we are interested in knowing, for each seller, the average number of bids received on all of his auctions in the last 5 days. This query requires a sliding-window join between $S_1$ and $S_2$ with a window on $S_1$ equal to the maximum lifetime of an auction, followed by an aggregation operator with a 5-day window.

Suppose memory is insufficient to retain all the tuples in $S_1$’s window, and suppose we use the frequency-based model for making load-shedding decisions in this scenario. Auction-ids are unique, so on stream $S_1$ we see each auction-id only once. On stream $S_2$, the arriving auction-ids are the currently open auctions, so this set changes over time. Thus, no fixed frequency distribution can be inferred through profiling. In this case, load-shedding schemes based on the frequency model [45] will simply retain new tuples and...
discard old ones. However, that is exactly the wrong thing to do, since most bids are received on auctions that are about to close, i.e., are relatively old. To capture this general type of scenarios, we propose a new *age-based* model defined for the sliding-window join $S_1[W_1] owtie S_2[W_2]$ as follows:

**Definition 3.2.2 (Age-Based Model).** For a tuple $s \in S_i$, $i = 1, 2$, the $S_i$-probe join tuples produced by $s$ obey the following two conditions:

1. The number of $S_i$-probe join tuples produced by $s$ is a constant independent of $s$, and is denoted by $n_i$.

2. Out of the $n_i$ $S_i$-probe join tuples of $s$, $p_i(k)$ are produced when $s$ is between age $k - 1$ and $k$.

Define $C_i(k)$, as the cumulative number of $S_i$-probe join tuples that a tuple $s \in S_i$ produces by age $k$, i.e., $C_i(k) = \sum_{j=1}^{k} p_i(j)$.

Thus, according to this model, the number of joins a tuple produces is independent of its join-attribute value, but is a function of the age of the tuple in the window. Assumption 1 in Definition 3.2.2 is not strictly necessary for our approach. However, in the scenarios we have considered, the set of join-attribute values changes over time. Thus, even if $n_i(s)$ depends on $s$, a for a tuple $s \in S_i$, it would be difficult to infer this dependence by profiling the stream.

**Different Age Curves:** Consider a curve that plots $p_i(k)$ against $k$; we call this the *age curve* for window $S_i[W_i]$. Intuitively, the age curve shows how likely a tuple in $S_i[W_i]$ is to produce join tuples, as it becomes older. Different applications that adhere to the age-based model may have very different age-curve shapes:

- *Increasing:* An example is the auction scenario described above. In a typical auction, relatively few bids are received in the beginning, followed by a large number of bids when the auction is about to close. Thus $p_1(k)$ is small for small $k$, and increases with $k$ until the auction lifetime, after which it drops to 0.

- *Decreasing:* Consider a join between an Orders and a Fulfillments stream on order-id, with a window on the orders stream. Most parts of an order are fulfilled soon, but some may require backorder and are fulfilled later. Thus we expect to see a decreasing age curve.
• **Bell:** Consider a join between streams of readings from two different sensors, where two readings join if the difference between their timestamps is less than a certain threshold. This join may be used to discover correlations between readings from two different observation points taken at roughly the same time. In this case, the age curve is expected to be bell-shaped. The age $k$ at which the peak of the age curve occurs will be determined by factors such as clock skew between the two sensors, and the difference in network latency from the sensors to the stream system. We report on an experiment we performed of this form in Section 3.6.

### 3.2.3 Parameter Estimation

For using any of the models described above, the model parameters must be instantiated, i.e., we must determine the frequencies of occurrence $f_1(v)$ and $f_2(v)$ of values $v \in D$ for the frequency-based model, and the age curves for the age-based model. We assume the standard technique of using the past to predict the future, so parameters are estimated by profiling the streams. There is previous work on building histograms in an online fashion using small space [65, 66, 70], which can be used to estimate the values of $f_1(v)$ and $f_2(v)$. For the age-based model, $n_i, i = 1, 2,$ is estimated as the average number of $S_i$-probe join tuples that an $S_i$-tuple produces in its lifetime. Similarly, $p_i(k)$ is estimated as the average number of $S_i$-probe join tuples that an $S_i$-tuple produces between age $k - 1$ and $k$.

We do not need to collect $p_i(k)$ for each time unit $k$, but can use a coarser time granularity. To accurately determine $p_i(k)$, we should execute the join with the full window $S_i[W_i]$ being retained. For now, we assume that we can allocate an extra chunk of “monitoring memory” that is circulated periodically to each window in turn to monitor its parameters accurately. If this memory is not available, $p_i(k)$ can be approximately estimated by retaining a small fraction of the tuples on $S_i$ in $S_i[W_i]$ for their entire lifetime.

### 3.3 Max-Subset Approximation

Consider a sliding window join. When we perform load shedding due to insufficient memory, only a fraction of the true join result will actually be produced. We denote the fraction
of the result tuples produced as \textit{recall}.

\[ \text{Recall}(t) = \frac{\text{Number of result tuples produced up to time } t}{\text{Number of actual result tuples up to time } t} \]

\textbf{Definition 3.3.1 (Max-Subset Problem).} Given a fixed amount of memory for a sliding-window join \( S_1[W_1] \bowtie_A S_2[W_2] \), devise an online load-shedding strategy that maximizes \( \lim_{t \to \infty} \text{Recall}(t) \).

We first state a result on the hardness of the problem for arbitrary streams (Section 3.3.1), then present a load-shedding strategy for the age-based model (Section 3.3.2), and finally discuss the max-subset problem for the frequency-based model (Section 3.3.3).

\subsection*{3.3.1 Hardness Result}

A load-shedding strategy is \textit{optimal} if it eventually produces the maximum recall among all strategies using the same amount of memory. For bounded streams, an \textit{offline} strategy is one that is allowed to make its load-shedding decisions after knowing all the tuples that are going to arrive in the future. We show that for arbitrary streams, it is not possible for any online strategy to be \textit{competitive} with the optimal offline strategy.

Let \( S \) denote a bounded sequence of tuple arrivals on the streams \( S_1 \) and \( S_2 \). Consider any online strategy. Let \( R_{\text{on}}(M, S) \) denote the recall obtained at the end of executing the online strategy with memory \( M \) on the sequence \( S \). Similarly, let \( R_{\text{off}}(M, S) \) denote the recall for the optimal offline strategy. We assume \( M \) is insufficient to retain the entire windows. The online strategy is \( k \)-\textit{competitive} if for any sequence \( S \), \( R_{\text{off}}(M, S)/R_{\text{on}}(M, S) \leq k \).

\textbf{Theorem 3.3.2.} For the max-subset problem, no online strategy (even randomized) can be \( k \)-competitive for any \( k \) that is independent of the length of the input sequence.

\textit{Proof.} Consider a sliding-window join \( S_1[W_1] \bowtie S_2 \) with a window on \( S_1 \) only. Consider the following input:

- First, \( n \) tuples (\( n > M \)) with distinct join-attribute values \((v_1, v_2, \ldots, v_n)\) arrive on \( S_1 \).
- Next, \( n \) tuples arrive on \( S_2 \) each having the same join-attribute value \( v_i \), where \( i \) is chosen randomly from 1 to \( n \).
Consider any deterministic strategy. It has retained $M$ tuples out of the $n$ which arrived on $S_1$. Thus, the probability that $v_i$ is one of the values which was retained, is $M/n$. In this case, the result produced is of size $n$. In other cases, i.e. when $v_i$ is one of the values which was not retained, the result is of size 0. Hence the expected result size produced by a deterministic strategy is $M$.

However, the optimal offline strategy (since it knows the future) will retain $v_i$, and always produce a result of size $n$. Thus the expected competitive ratio of a deterministic strategy on this input is $n/M$. By Yao’s min-max technique, this is a lower bound on the competitive ratio of any online strategy (possibly randomized). By increasing $n$, the competitive ratio can be made worse, hence the result follows.

This result shows that we cannot expect to find an effective load-shedding strategy that addresses the max-subset problem for arbitrary streams.

### 3.3.2 Age-Based Model

Consider the max-subset problem for a join $S_1[W_1] \bowtie S_2[W_2]$ that adheres to the age-based model. We first assume a fixed amount of memory is available for $S_1[W_1]$, and consider the problem of maximizing the number of $S_1$-probe join tuples produced. A symmetric procedure applies for maximizing the number of $S_2$-probe join tuples given a fixed memory for $S_2[W_2]$. Then we show how to allocate the overall available memory between $S_1[W_1]$ and $S_2[W_2]$ to maximize the recall of the entire join.

#### 3.3.2.1 Fixed Memory for $S_1[W_1]$

Suppose the available memory for $S_1[W_1]$ is sufficient to store $M_1$ tuples of stream $S_1$. We denote the amount of memory required to store one tuple as a “cell”. For now we assume $r_1 = 1$, i.e., one tuple arrives on $S_1$ at each time step. At the end of the section we show the easy generalization to other $r_1$.

We first give the optimal strategy for $M_1 = 1$, which forms the building block for our strategy for $M_1 > 1$. Recall that $C_1(k)$ denotes the total number of $S_1$-probe join tuples that a tuple $s \in S_1$ produces by age $k$. Let $k_1^{opt}$ denote the $k (~\leq W_1)$ at which $\frac{C_1(k)}{k}$ is maximized.

---

$^1$Note that all of our optimality claims assume constant rather than average $r_1$, however our experiments (Section 3.6) show that our algorithm performs well for a distribution of arrival rates.
Strategy 3.3.3 ($M_1 = 1$). Retain the first tuple $s \in S_1$ in $S_1[W_1]$ for $k_1^{opt}$ time units, discarding other tuple arrivals on $S_1$. After $k_1^{opt}$ time units, discard $s$, retain the tuple arriving next for the next $k_1^{opt}$ time units, and continue.

Proof. Suppose the optimal strategy picks the first tuple, and keeps it for $k$ time units before discarding it. According to the age-based model, the new tuple picked up is identical in terms of producing $S_1$-probe join tuples. Thus, by symmetry, it too must be retained for $k$ time units. Thus, every $k$ time units, $C_1(k)$ result tuples are produced. To maximize the number of result tuples, $k$ should be chosen such that $C_1(k)/k$ is maximized.

Example 3.3.4. Let $r_1 = 1$ and $M_1 = 1$ as we have assumed so far. Let the window size $W_1 = 4$, and let the age curve be defined by $p_1(1) = 1, p_1(2) = 1, p_1(3) = 2, p_1(4) = 1$. $C_1(k)/k$ is maximized at $k_1^{opt} = 3$.

Let $s_i$ denote the tuple arriving at time $i$ on $S_1$. The following diagram illustrates Strategy 3.3.3 on this example. Entries in the third row denote the number of $S_1$-probe join tuples produced between each time step and the next.

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell 1</td>
<td>$s_1$</td>
<td>$s_1$</td>
<td>$s_4$</td>
<td>$s_4$</td>
<td>$s_4$</td>
<td>$s_7$</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>Discard</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_1$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_4$</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td># Results</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>. . .</td>
</tr>
</tbody>
</table>

Strategy 3.3.3 produces 4 join tuples every 3 time units and is optimal among all possible strategies.

Now suppose $M_1 > 1$. We must consider two cases:

1. If $k_1^{opt} \geq M_1$, the optimal strategy is to run Strategy 3.3.3 “staggered”, for each of the $M_1$ cells. For example, if $M_1 = 2$ in Example 3.3.4, we get:

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell 1</td>
<td>$s_1$</td>
<td>$s_1$</td>
<td>$s_4$</td>
<td>$s_4$</td>
<td>$s_4$</td>
<td>$s_7$</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>Cell 2</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_5$</td>
<td>$s_5$</td>
<td>$s_5$</td>
<td>$s_5$</td>
<td>. . .</td>
<td></td>
</tr>
<tr>
<td>Discard</td>
<td>$s_3$</td>
<td>$s_1$</td>
<td>$s_2$</td>
<td>$s_6$</td>
<td>$s_4$</td>
<td>. . .</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Results</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>. . .</td>
<td></td>
</tr>
</tbody>
</table>

2. If $k_1^{opt} < M_1$, the problem becomes more complex because running a staggered Strategy 3.3.3 uses only $k_1^{opt}$ cells, thereby underutilizing the available memory.
To address Case 2 ($k_{opt}^1 < M_1$), we first define an age curve with a minima. The age curve $p_1(k)$ against $k$ has a minima if there exist $k_1 < k_2 < k_3$ such that $p_1(k_1) > p_1(k_2)$ and $p_1(k_2) < p_1(k_3)$.

If the age curve has no minima, the optimal strategy is to retain every tuple for exactly $M_1$ time units. Once a tuple has been retained for $k_{opt}^1$ time units, retaining it any further becomes less useful, and since the curve has no minima the tuple cannot become more useful in the future. Thus, it should be discarded as early as possible after $k_{opt}^1$ time units. At the same time, tuples should not be discarded any earlier than $M_1$ time units, as that would lead to underutilization of memory.

If the age curve has a minima, retaining each tuple for exactly $M_1$ time units may be suboptimal. We illustrate the subtleties through an example.

Example 3.3.5. Let $W_1 = 3$ and $M_1 = 2$. Let the age curve be defined by $p_1(1) = 3$, $p_1(2) = 0$, and $p_1(3) = 2$. Thus, the age curve has a minima at $k = 2$. We have $k_{opt}^1 = 1$, so $k_{opt}^1 < M_1$. The following strategy alternates between retaining every tuple for 1 and 3 time units, and by exhaustive search is seen to be optimal for this example:

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell 1</td>
<td>$s_1$</td>
<td>$s_1$</td>
<td>$s_1$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_5$</td>
<td>$s_5$</td>
<td>...</td>
</tr>
<tr>
<td>Cell 2</td>
<td>$s_2$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_6$</td>
<td>$s_6$</td>
<td>$s_7$</td>
<td>...</td>
</tr>
<tr>
<td>Discard</td>
<td>$s_2$</td>
<td>$s_1$</td>
<td>$s_4$</td>
<td>$s_3$</td>
<td>$s_6$</td>
<td>$s_3$</td>
<td>$s_6$</td>
<td>...</td>
</tr>
<tr>
<td># Results</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>...</td>
</tr>
</tbody>
</table>

This strategy produces an average of 4 join tuples per time unit. Note that retaining every tuple for $M_1 = 2$ time units produces only 3 join tuples per time unit.

We do not have an optimal strategy for the general case of age curves with minima, but in practice, age curves are unlikely to have minima (e.g., none of the examples discussed in Section 3.2.2 have minima). However, for completeness, we give the following greedy heuristic for this case. For each tuple $s \in S_1[W_1]$, assign a priority that represents the fastest rate at which $s$ can produce $S_1$-probe join tuples. The priority of a tuple at age $i$ is given by:

$$Priority(i) = \max_{i < j \leq W_1} \frac{C_1(j) - C_1(i)}{j - i}$$

When a tuple needs to be discarded due to a memory constraint, the tuple with the lowest priority is discarded.
3.3. MAX-SUBSET APPROXIMATION

This greedy strategy leads to the optimal solution for Example 3.3.5. Interestingly, this strategy reduces to the optimal strategy for all the previous cases as well. We do not further consider age curves with minima.

We shall refer to the overall approach for the age-based max-subset problem presented in this section as the AGE algorithm. We evaluate AGE experimentally in Section 3.6.

3.3.2.2 Fixed Memory for \( S_1[W_1] + S_2[W_2] \)

So far we have addressed the problem of maximizing the number of \( S_i \)-probe join tuples, \( i = 1, 2 \), given a fixed amount of memory for \( S_i[W_i] \). Now suppose we have a fixed amount of memory \( M \) for the entire join. To determine how to allocate the available memory between \( S_1[W_1] \) and \( S_2[W_2] \), we need a function that relates the memory allocation to the overall recall obtained. Let \( M_i \) be the memory allocated to \( S_i[W_i] \). Let \( R_i \) denote the rate at which \( S_i \)-probe join tuples are produced. If the AGE algorithm from Section 3.3.2.1 is applied:

\[
R_i = \begin{cases} 
M_i \frac{C_i(k_i^{\text{opt}})}{k_i^{\text{opt}}} & \text{if } M_i \leq k_i^{\text{opt}} \\
C_i(M_i) & \text{if } M_i > k_i^{\text{opt}} 
\end{cases} \tag{3.2}
\]

Then the overall recall of the join is given by \( \frac{R_1 + R_2}{r_1n_1 + r_2n_2} \). To determine the memory allocation between \( S_1[W_1] \) and \( S_2[W_2] \), we simply find \( M_1 \) and \( M_2 \) such that this expression for the recall of the join is maximized, subject to the constraint \( M_1 + M_2 = M \).

Finally, so far we have assumed \( r_i = 1 \). If \( r_i > 1 \), the system can be viewed as \( r_i \) systems running in parallel, each having \( M_i/r_i \) memory, and having unit arrival rate. This equivalence exists because for the age-based model, each incoming tuple on \( S_1 \) is identical in terms of producing \( S_1 \)-probe join tuples, Thus, if \( r_i > 1 \) and memory \( M_i \) is available for \( S_i[W_i] \), Equation 3.2 becomes:

\[
R_i = \begin{cases} 
M_i \cdot \frac{C_i(k_i^{\text{opt}})}{k_i^{\text{opt}}} & \text{if } M_i/r_i \leq k_i^{\text{opt}} \\
r_i \cdot C_i(M_i/r_i) & \text{if } M_i/r_i > k_i^{\text{opt}} 
\end{cases} \tag{3.3}
\]

The recall for the entire join is then given by \( \frac{R_1 + R_2}{r_1n_1 + r_2n_2} \).
3.3.3 Frequency-Based Model

We briefly consider the max-subset problem for the frequency-based model as covered in [45]. We derive the recall obtained given a fixed amount of memory for the join. This relationship between memory and recall is needed in Section 3.5 for overall memory allocation across joins.

Consider $S_1$-probe join tuples first. Recall Definition 3.2.1 of the frequency-based model. The following approach, called PROB, is suggested in [45]: Every tuple $s_1 \in S_1[W_1]$ is assigned a priority equal to $f_2(s_1.A)$. If a tuple needs to be discarded due to a memory constraint, the tuple with the lowest priority is discarded. Without loss of generality, assume the values in $D$ are $v_1, \ldots, v_n$, and for $i < j$, $f_2(v_i) \geq f_2(v_j)$. Then for $i < j$, PROB will prefer to retain all instances of $v_i$ in $S_1[W_1]$ over any instance of $v_j$. Let $M_1$ be the memory allocated to $S_1[W_1]$. PROB will retain all instances of $v_1, v_2, \ldots, v_i$, where $i$ is the largest number such that $r_1W_1 \sum_{j=1}^{i} f_1(v_j) \leq M_1$. (A fraction of the instances of $v_{i+1}$ will be retained too, but our analysis is not affected significantly.) Thus, $S_1$-probe result tuples are produced at a rate given by $R_1 = r_1r_2W_1 \sum_{j=1}^{i} f_1(v_j) f_2(v_j)$. A symmetric expression can be derived for the rate $R_2$ at which the $S_2$-probe join tuples are produced, given memory $M_2$ for $S_2[W_2]$. The overall recall of the join is then given by $\frac{R_1 + R_2}{r_1r_2(\sum_{v \in D} f_1(v)f_2(v))}$. Thus, given a total amount of memory $M$ for the join, we can find $M_1$ and $M_2$ such that the overall recall of the join is maximized, subject to the constraint $M_1 + M_2 = M$.

3.4 Random Sampling

In this section, we address the problem of extracting a random sample of the $S_1[W_1] \bowtie_A S_2[W_2]$ join result with limited memory. We first state a result on the hardness of performing uniform random sampling on the join result for arbitrary streams (Section 3.4.1). We then give an algorithm for uniform random sampling that applies for both the age-based and frequency-based models (Section 3.4.2). Finally, in Section 3.4.3, we consider the case when a uniform sample is not required directly by the application, but is being gathered only for estimating an aggregate over the join result. For these cases, we consider a statistically weaker form of sampling called cluster sampling [40], which can be performed more easily than uniform sampling, and often yields a more accurate estimate of the aggregate.
3.4. RANDOM SAMPLING

3.4.1 Hardness Result

For sampling over the windowed join result of arbitrary streams, we have the following negative result:

**Theorem 3.4.1.** If the available memory is insufficient to retain the entire windows, it is not possible to guarantee a uniform random sample for any sampling fraction \( p > 0 \).

*Proof.* Suppose we are uniformly sampling a fraction \( p > 0 \) of the join result. Consider a tuple \( s_1 \in S_1[W_1] \). Let \( n_1(s_1) \) denote the number of \( S_1 \)-probe joins that \( s_1 \) participates in, during its lifetime. Suppose \( s_1 \) has already participated in \( n \) \( S_1 \)-probe joins. \( s_1 \) can be discarded from \( S_1[W_1] \) only if it is guaranteed that none of its remaining \( n_1(s_1) - n \) join tuples are going to be included in the sample. This happens with probability \( (1 - p)^{n_1(s_1) - n} \).

Since the tuple arrival is arbitrary, there is no upper bound available for \( n_1(s_1) \), and the above probability can be brought arbitrarily close to 0, for any \( p > 0 \). Thus, no tuple can be discarded from \( S_1[W_1] \) earlier than it expires, and the required memory is \( M_{\text{req}} \). If \( M < M_{\text{req}} \), it follows that some tuples must be discarded earlier, and the semantics of sampling cannot be preserved in general.

The above result shows that we cannot expect to find an effective procedure that performs uniform random sampling over the join result of arbitrary streams with limited memory. However, we can compute a sample when we have a model of stream arrivals, as we show next.

3.4.2 Uniform Random Sampling

For random sampling we can consider the frequency-based and the age-based models together. We shall assume *Bernoulli sampling*, or sampling under the coin-flip semantics [34]: for sampling a fraction \( p \) from a set of tuples, every tuple in the set is included in the sample with probability \( p \) independent of every other tuple.

3.4.2.1 Sampling Algorithm

Our algorithm *UNIFORM* for uniform random sampling over a sliding-window join with limited memory is shown in Figure 3.3. We only show the procedure for sampling from the \( S_1 \)-probe join tuples by selectively retaining tuples in \( S_1[W_1] \). The procedure for sampling from the \( S_2 \)-probe join tuples is analogous. *UNIFORM* needs to know, for each arriving
CHAPTER 3. MEMORY-LIMITED EXECUTION OF WINDOWED STREAM JOINS

$s_1$ : Tuple arriving on $S_1$

$n_1(s_1)$ : Number of $S_1$-probe join tuples that $s_1$ produces

$p$ : Sampling fraction

$\text{DecideNextJoin}(s_1)$:
1. pick $X \sim G(p)$
2. $s_1.n_{ext} = s_1.n_{um} + X$
3. if $(s_1.n_{ext} > n_1(s_1))$
4. discard $s_1$

$\text{Join}(s_1, s_2)$:
1. $s_1.n_{um} = s_1.n_{um} + 1$
2. if $(s_1.n_{um} = s_1.n_{ext})$
3. output $s_1 \bowtie_A s_2$
4. $\text{DecideNextJoin}(s_1)$

Figure 3.3: Algorithm $\text{UNIFORM}$

When a tuple $s_1$ arrives on $S_1$, $s_1.n_{um}$ is initialized to 0, and the procedure $\text{DecideNextJoin}(s_1)$ is called. $\text{Join}(s_1, s_2)$ is called when a tuple $s_2$, that joins with $s_1$, arrives on $S_2$. $G(p)$ denotes the geometric distribution with parameter $p$ [97], and $X \sim G(p)$ denotes that we pick $X$ at random from $G(p)$. When $\text{DecideNextJoin}(s_1)$ is called, $\text{UNIFORM}$ logically flips coins with bias $p$ for deciding the next $S_1$-probe join tuple of $s_1$ that will be included in the sample. If all remaining $S_1$-probe join tuples of $s_1$ are rejected by the coin flips, $s_1$ is discarded, thus enabling memory reduction.

3.4.2.2 Determining the Sampling Fraction $p$

To determine the sampling fraction $p$, we first obtain the expected memory usage of $\text{UNIFORM}$ (i.e., the expected number of tuples retained) in terms of $p$. We then equate this expected memory usage to the amount of memory available for performing the join and solve for $p$. For robustness, we can also calculate the variance of the memory usage of $\text{UNIFORM}$ and decide the sampling fraction such that the probability of the memory usage exceeding the available memory is sufficiently small. Note that now the tuple size must include the space required to store the extra fields $n_{ext}$ and $n_{um}$ (Figure 3.3).

For a given sampling fraction $p$, we now derive the expected memory usage of $\text{UNIFORM}$ for both the age-based and frequency-based models.
3.4. RANDOM SAMPLING

**Age-Based Model:** Recall Definition 3.2.2. Recall that $C_1(k)$ denotes the cumulative number of $S_1$-probe join tuples that a tuple $s_1 \in S_1$ produces by age $k$. Define the inverse of the $C_1$ function, $C^{-1}_1(m)$, as the smallest $k$ such that $C_1(k) \geq m$. Thus, a tuple $s_1 \in S_1$ produces $m$ $S_1$-probe join tuples by the time its age is $C^{-1}_1(m)$.

**Theorem 3.4.2.** For the age-based model, the expected memory usage of $S_1[W_1]$ is

$$r_1 \sum_{i=1}^{n_1} p(1 - p)^{n_1-i}C^{-1}_1(i).$$

**Proof.** According to the sampling scheme, a tuple $s_1$ is discarded after participating in exactly $i$ joins ($i \geq 1$), if the $i^{th}$ join is included, and its remaining $n_1 - i$ joins are not included in the sample. This happens with probability $p(1 - p)^{n_1-i}$. Thus the expected age after which a tuple is discarded is given by:

$$t_{dis} = \sum_{i=1}^{n_1} p(1 - p)^{n_1-i}C^{-1}_1(i)$$

Since tuples are arriving on $S_1$ at the rate of $r_1$ tuples per time unit, the expected size of $S_1[W_1]$ is $r_1t_{dis}$. □

**Frequency-Based Model:** Recall Definition 3.2.1. We assume that the $S_1$-probe join tuples of a tuple $s_1 \in S_1$ are produced uniformly throughout the lifetime of $s_1$ (because a uniform fixed fraction of tuples arriving on $S_2$ join with $s_1$).

In the frequency-based model, the expected age after which a tuple is discarded by the sampling procedure is only a function of its join-attribute value. Let $t_{dis}(v)$ denote the expected age after which a tuple with join-attribute value $v$ is discarded by the sampling procedure.

**Lemma 3.4.3.** For sampling fraction $p$,

$$t_{dis}(v) = \left(1 - \frac{q(1 - q^r_2W_1f_2(v))}{pr_2W_1f_2(v)}\right) \cdot W_1$$

where $q = 1 - p$. □

**Proof.** The probability that a tuple $s_1$ is discarded after participating in exactly $i$ joins is as in the proof of Theorem 3.4.2. In the frequency model, it can be assumed that the $S_1$-probe joins of $s_1$ are uniformly spread throughout the lifetime of $s_1$ (because a uniform
fixed fraction of tuples arriving on $S_2$ join with $s_1$). Thus, the expected age after which $s_1$ is discarded is given by:

\[
\sum_{i=1}^{n_1(s_1)} pq^{n_1(s_1)-i} \cdot \frac{i}{n_1(s_1)} \cdot W_1
\]

Suppose $s_1.A = v$. The result follows by putting $n_1(s_1) = r_2W_1f_2(v)$ (by Equation 3.1), and performing the summation.

**Theorem 3.4.4.** For the frequency-based model, the expected memory usage of $S_1[W_1]$ is:

\[
r_1W_1 \sum_{v \in D} f_1(v) \left(1 - \frac{q(1 - q^{r_2W_1f_2(v)})}{pr_2W_1f_2(v)}\right)
\]

where $q = 1 - p$.

**Proof.** Since tuples are arriving on $S_1$ at the rate of $r_1$ tuples per time unit, and a fraction $f_1(v)$ of these have value $v$ in the join attribute, the expected size of $S_1[W_1]$ is $r_1 \sum_{v \in D} f_1(v)t_{dis}(v)$. The result follows on substituting for $t_{dis}(v)$ from Lemma 3.4.3.

In both models, a symmetric expression holds for the expected memory usage of $S_2[W_2]$, assuming we use the same sampling fraction $p$ for the $S_2$-probe join tuples. Summing these expressions gives us the total memory usage for the join $S_1[W_1] \bowtie_A S_2[W_2]$.

### 3.4.3 Cluster Sampling

The correctness of UNIFORM depends heavily on the accuracy with which $n_i(s)$ is estimated for a tuple $s \in S_i$, $i = 1, 2$. For example, for a tuple $s_1 \in S_1$, if $n_1(s_1)$ is underestimated as $n'_1(s_1)$, then all the $S_1$-probe join tuples of $s_1$ subsequent to its first $n'_1(s_1)$ join tuples will never be selected for the sample. On the other hand, if $n_1(s_1)$ is overestimated, $s_1$ may remain in $S_1[W_1]$ until expiry, waiting for joins that never take place, and the overall memory usage may be considerably higher than the expected value derived in Theorems 3.4.4 and 3.4.2.

If a uniform random sample of the join is not required directly by the application, but the sample is being taken only to estimate an aggregate over the join results, these difficulties can be overcome by using a statistically weaker form of sampling called *cluster sampling* [40].
In general, cluster sampling is applicable when the population to be sampled can be divided into groups, or clusters, such that the cost of sampling a single element of a cluster is equal to that of sampling the entire cluster. Thus, for cluster sampling, a certain number of clusters are chosen at random, and all elements of selected clusters are included in the cluster sample. A cluster sample is unbiased, i.e., each element of the population has equal probability of being included in the sample. However, it is correlated, i.e., the inclusion of tuples is not independent of each other as in a uniform sample. A detailed analysis of cluster sampling can be found in [40]. In the remainder of this section we assume the sample of the join is being gathered for estimating either a sum or an average aggregate, and the objective is to minimize the error in the estimated aggregate. If a cluster sample is used for aggregation, the accuracy of the estimate depends on how the value of the aggregate varies across clusters. If inter-cluster variance is small, the estimate obtained is accurate, but accuracy degrades with increasing inter-cluster variance.

### 3.4.3.1 Two Approaches

Consider sampling from the \( S_1 \)-probe join tuples; a symmetric procedure applies for sampling from the \( S_2 \)-probe join tuples. A tuple \( s_1 \in S_1 \) joins with \( n_1(s_1) \) tuples arriving on \( S_2 \). These join tuples form a cluster, and the entire cluster can be sampled by simply retaining \( s_1 \) in \( S_1[W_1] \) until expiry. The fraction of clusters that can be sampled is determined by the number of tuples that can be retained until expiry in the memory available for \( S_1[W_1] \). Thus we have the following naïve approach to cluster sampling.

**Strategy 3.4.5 (EQ-CLUSTER).** Add an incoming tuple \( s_1 \in S_1 \) to \( S_1[W_1] \) with probability \( p \). If \( s_1 \) is added to \( S_1[W_1] \), retain it until expiry and include all its \( S_1 \)-probe join tuples in the sample.

Notice that this scheme does not depend on \( n_1(s_1) \), and always produces an unbiased sample. The expected memory usage for \( S_1[W_1] \) according to this scheme is \( r_1 W_1 p \). Thus, \( p \) can be decided based on the amount of memory available.

**EQ-CLUSTER** is suitable when the clusters are roughly of equal size (e.g., as in the age-based model). However, if clusters are of unequal sizes, as in the frequency-based model, statistics literature [40] suggests that better estimates of the aggregate can be obtained by selecting a cluster with probability proportional to its size. Otherwise, if clusters are selected with equal probability, large clusters that contribute most to the aggregate may be
missed altogether. We thus have the following approach:

**Strategy 3.4.6 (PPS-CLUSTER).** Add an incoming tuple \( s_1 \in S_1 \) to \( S_1[\tilde{W}_1] \) with probability proportional to \( n_1(s_1) \). If \( s_1 \) is added to \( S_1[\tilde{W}_1] \), retain it until expiry and include all its \( S_1 \)-probe join tuples in the sample.

With PPS-CLUSTER, to get an unbiased estimate of the aggregate, we must perform weighted aggregation on the cluster sample: the contribution of each cluster to the aggregate is assigned a weight inversely proportional to the cluster size. Details can be found in [40]. Notice that even if \( n_1(s_1) \) is incorrectly estimated, the same incorrect estimate is used in performing weighted aggregation. Hence, the resulting estimate of the aggregate is still unbiased.

Consider the application of PPS-CLUSTER for the frequency-based model. Since \( n_1(s_1) \propto f_2(s_1.A) \), let \( s_1 \) be added to \( S_1[\tilde{W}_1] \) with probability \( p \cdot f_2(s_1.A) \) where \( p \) is a proportionality constant. The expected memory usage of \( S_1[\tilde{W}_1] \) is \( r_1 W_1 p \sum_{v \in \mathcal{D}} f_1(v) f_2(v) \). Thus, \( p \) can be determined according to the amount of memory available.\(^2\)

### 3.4.3.2 Comparison of Approaches

To summarize, let us briefly consider which sampling approach is preferable in different scenarios. Recall that the objective is to minimize the error in an estimated aggregate. The relevant factors to be considered are:

- **Accuracy of model parameters:** If \( n_i(s) \) is incorrectly estimated for a tuple \( s \in S_i \), \( i = 1, 2 \), UNIFORM may perform poorly since it may produce a biased sample. In this case, cluster sampling should be used.

- **Inter-cluster variance:** Consider the variance in the values of the aggregate for different clusters. The lower this variance, the better the performance of cluster sampling compared to uniform sampling [40].

- **Cluster sizes:** PPS-CLUSTER should be used for unequal-size clusters. PPS-CLUSTER reduces to EQ-CLUSTER for equal-size clusters.

\(^2\)A value of \( p \) obtained in this way can cause \( p f_2(s_1.A) \) to exceed 1 for some \( s_1 \), resulting in an overestimate of memory usage. The correct value of \( p \) can be chosen by an iterative procedure, where in each iteration, we truncate probabilities > 1 to be = 1.
3.5 Memory Allocation across Multiple Joins

Now suppose our stream system is executing a number of continuous queries, each of which involves a sliding-window join. Even after exploiting sharing for windows on the same stream, there still remains the question of how memory should be allocated across the windows for different streams. In this section, we assume all joins involve different streams and address the problem of allocating the available memory across multiple joins.

For now, let us assume the unweighted case, i.e., all joins are equally important. The goal of our allocation scheme is to ensure that no join does “too badly” in terms of approximation error, i.e., we seek to minimize the maximum approximation error in any join. It is important to observe that different joins may differ in the accuracy of their approximation even when given the same fraction of their memory requirement. Thus, simple proportional allocation of memory among the joins is generally not optimal.

Suppose there are \( n \) sliding-window joins with an overall memory constraint \( M \). Each join may follow either the age-based or the frequency-based model. Further, each join has a certain approximation metric which we denote by \( Q \): For the max-subset problem, \( Q \) is the recall of the join. For the sampling problem, \( Q \) is the error in an aggregate (e.g., SUM) estimated from the sample. We assume that each join uses the same approximation metric (i.e., either recall or aggregation error), otherwise the choice of a combined approximation metric is not clear. We shall focus on the case when \( Q \) is recall. A similar technique applies when \( Q \) is aggregation error.

For a particular memory allocation, let \( q_i \) be the recall obtained for the \( i \)th join. The optimal memory allocation we seek is the one that maximizes \( \min_{1 \leq i \leq n} q_i \). The key to our scheme is the following observation (a similar observation is made in [20]).

**Theorem 3.5.1.** To maximize the minimum recall, the optimal memory allocation is one that produces the same recall in all joins.

By Theorem 3.5.1, in the optimal memory allocation the recall obtained in each join is the same, say \( q_{opt} \). Let \( f_i(q) \) denote the minimum amount of memory required to obtain recall \( q \) in the \( i \)th join. Then \( q_{opt} \) is the maximum \( q \) such that \( \sum_{i=1}^{n} f_i(q) \leq M \). Assuming the functions \( f_i \) are known, \( q_{opt} \) can be found by an iterative binary search. The amount of memory to be allocated to the \( i \)th join is then given by \( f_i(q_{opt}) \).

Let us consider how the function \( f_i(q) \) can be obtained for the \( i \)th join. Recall that we specified the relationship between memory available for a join and the recall obtained,
both for the age-based (Section 3.3.2.2) and the frequency-based (Section 3.3.3) models. These can be used to obtain \( f_i(q) \). When the metric \( Q \) is aggregation error, we use the relationship between memory and sampling fraction (Theorems 3.4.4 and 3.4.2). The expected aggregation error for a given sampling fraction can be derived in terms of population characteristics such as mean and variance [20]. Together, these can be used to calculate \( f_i(q) \).

Finally, suppose that different joins have different relative importance. Let \( w_i \) be the weight of the \( i^{th} \) join. Now our objective is to maximize \( \min_{1 \leq i \leq n} q_i/w_i \). Our argument extends to show that the optimal solution is to allocate memory \( f_i(w_iq_{opt}) \) to the \( i^{th} \) join, where \( q_{opt} \) is the maximum \( q \) such that \( \sum_{i=1}^{n} f_i(w_iq) \leq M \).

We shall refer to the approach for memory allocation presented in this section as \( ALLOC \), and evaluate its performance experimentally in Section 3.6.

### 3.6 Experiments

We now present an experimental evaluation of our techniques. Our experiments demonstrate the following:

1. In a real-life scenario that adheres to the age-based model, our algorithm \( AGE \) (Section 3.3.2.1) gives considerably higher recall than more naïve approaches.

2. Our sampling approaches \( UNIFORM \) and \( PPS-CLUSTER \) (Section 3.4) provide low-error estimates of windowed aggregates over the join result. Either of the two approaches may be preferable, depending on the specific scenario.

3. Our algorithm \( ALLOC \) for memory allocation across joins (Section 3.5) significantly outperforms simple proportional allocation in terms of maximizing the minimum recall.

#### 3.6.1 Age-Based Experiment

For initial experimentation with the age-based model, we captured real data as follows. We set up two stream sources, \( \phi_1 \) and \( \phi_2 \), and a central server. Source \( \phi_1 \) and the server run on the same physical machine, while source \( \phi_2 \) runs on a distant machine connected over a wide-area network (WAN). Each source produces tuples at a constant rate of \( r_1 = r_2 = 50 \) per second. Each tuple contains a timestamp \( ts \) from the local clock at the source. All tuples are streamed to the server using a UDP channel.
3.6. EXPERIMENTS

Denote the streams from sources $\phi_1$ and $\phi_2$ as $S_1$ and $S_2$ respectively. We execute a sliding-window join whose purpose is to identify causal correlation between the two streams—to do so, it matches tuples from $S_2$ with tuples from $S_1$ that were timestamped approximately one minute earlier. The join predicate chosen is $S_2.t_s - S_1.t_s \in [59.9, 60.1]$ where time units are seconds. To ensure that $S_1$ tuples do not expire before matching $S_2$ tuples arrive (the network latency from source $\phi_2$ to the server is high), we conservatively set the window on $S_1$ as $W_1 = 2$ minutes. Since joining tuples always arrive later on $S_2$ than on $S_1$, a window on $S_2$ need not be stored.

We generated a trace of approximately 40 minutes of tuple arrivals at the server and then used this trace for repeatability. Figure 3.4 shows the age curve ($p_1(k)$ vs. $k$) determined by an initial pass through our trace. We show $p_1(k)$ as a fraction of $n_1$ (recall Definition 3.2.2). The granularity chosen for $k$ was 0.1 second. We see that a tuple $s \in S_1$ produces most join tuples at an age of approximately $k = 63$ seconds. Out of this, a 60 second delay is due to the join predicate, and the rest of the delay is due to clock skew between sources $\phi_1$ and $\phi_2$, and significantly higher network latency for tuples from $\phi_2$ than from $\phi_1$. 

![Figure 3.4: Age curve for WAN experiment](image)
3.6.1.1 Results

Figure 3.5 shows the recall obtained on our trace by various load-shedding approaches as we vary the amount of allocated memory. Memory is shown as a percentage of the amount required to retain the entire window \( (r_1 W_1) \). We compare: (1) AGE: Section 3.3.2.1; (2) UNTIL-EXPIRY: A tuple is added to \( S_1[W_1] \) only if memory is available, and then retained until expiry; (3) RECENT: The most recent tuples in the window are retained; and (4) Theoretical-AGE: The recall that should be theoretically obtained by applying the AGE approach, as given by Equation 3.3. Note that RECENT is the approach that we get if we simply apply the frequency-based model in this scenario.

Although in reality the age curve shown in Figure 3.4 has some minima, \( p_1(k) \) never increases significantly after decreasing. Hence, for all practical purposes, we can apply our AGE approach assuming the curve has no minima. \( k_1^{opt} \) was calculated to be 68.8 seconds.

We see that AGE outperforms RECENT and UNTIL-EXPIRY. RECENT performs especially badly, producing no join tuples even when the allocated memory is as much as 40%. However, when the allocated memory is high enough so that \( M_1 \geq r_1 k_1^{opt} \), AGE reduces to RECENT (see Equation 3.3), and hence both approaches produce the same recall. Note that if \( W_1 \) had been conservatively set to be higher, the performance of UNTIL-EXPIRY would degrade, whereas the performance of AGE would not be affected. We also see that the actual recall obtained by AGE closely agrees with the theoretically predicted value.
3.6. EXPERIMENTS

3.6.2 Experiments on Synthetic Data

For the next set of experiments, we synthetically generate streams $S_1$ and $S_2$ for both the age-based and the frequency-based model, and perform the sliding-window join $S_1[W_1] \bowtie S_2[W_2]$ with limited memory. For simplicity, we consider only the $S_1$-probe join tuples in our experimental results. For both models, tuples on streams $S_i, i = 1, 2$, are generated at an average rate of $r_i$ tuples per unit time. This is done by choosing the inter-arrival time uniformly at random between $1/2r_i$ and $2/r_i$ time units. For all experiments we fix $r_1 = 1, r_2 = 5$, and $W_1 = 500$.

3.6.2.1 Age-Based Data Generation

First stream $S_1$ is generated. Each tuple on $S_1$ contains a unique id which serves as the join attribute, as in the examples of Section 3.2.2 (e.g., in the auction scenario, each tuple on $S_1$ has a unique auction-id). Next, we specify the age curve for $S_1[W_1]$ by dividing the window duration $W_1$ into $m$ buckets and specifying $p_1(k)$ for the $k^{th}$ bucket. The first bucket consists of the newest tuples, and the $m^{th}$ bucket the oldest tuples. We fix $n_1 = 5$ and $m = 20$.

We then generate stream $S_2$ according to this age curve. Suppose a tuple is to be generated on $S_2$ at time $t$. The value of its join attribute is determined as follows. We choose one of the $m$ buckets at random with the $k^{th}$ bucket being chosen with probability $p_1(k)/n_1$. Then, we choose one tuple at random from all the tuples of $S_1[W_1]$ occupying the chosen bucket at time $t$. The id of this randomly-chosen tuple is assigned as the join-attribute value of the newly generated tuple on $S_2$.

3.6.2.2 Max-Subset Problem with Age-Based Data

We experimented with three different age curves. (1) Increasing (INC): $p_1(k) \propto k^2$; (2) Decreasing (DEC): $p_1(k) \propto (m - k)^2$; and (3) Bell-shaped (BELL): $p_1(k) \propto k^2$ for $1 \leq k \leq m/2$ and $p_1(k) \propto (m - k)^2$ for $m/2 < k \leq m$. Figure 3.6 shows a comparison of the recall obtained by various approaches for different types of age curves. For the INC curve, AGE significantly outperforms RECENT. For the DEC curve, AGE reduces to RECENT, so we do not show their results separately. For the BELL curve, AGE outperforms RECENT until $M_1 < r_1k_1^{opt}$ (see Equation 3.3). For $M_1 \geq r_1k_1^{opt}$, AGE reduces to RECENT.

Note that for the same amount of allocated memory, the recall differs greatly depending
on the shape of the age curve. This indicates that in the presence of multiple joins, in order to maximize the minimum recall, simple proportional memory allocation is not sufficient, which we verify empirically in Section 3.6.2.5.

3.6.2.3 Frequency-Based Data Generation

Data generation for the frequency-based model is relatively easier than for the age-based model. We choose a domain $D$. The domain size is fixed at $|D| = 50$. For each stream, the join-attribute values are drawn from a Zipfian distribution of parameter $Z$ over $D$ [132]. The distribution used for both streams need not be the same. We consider three cases: (1) Directly Correlated (DC): The order of frequency of occurrence of values is the same for $S_1$ and $S_2$; (2) Inversely Correlated (IC): The order of frequency of occurrence of values for $S_1$ is opposite of that for $S_2$, i.e., the rarest value on $S_1$ is the most common on $S_2$ and vice-versa; and (3) Uncorrelated (UC): The order of frequency of occurrence of values for the two streams is uncorrelated.

3.6.2.4 Random Sampling

To evaluate our sampling approaches, we perform a windowed average over the sampled result of a join, and compare the approaches in terms of aggregation error. We report

Figure 3.6: Recall obtained on synthetic age-based data
results only for the case when the join follows the frequency-based model. Results for the age-based model are similar. The aggregation window is fixed at $W_{\text{aggr}} = 500$. The values of the aggregation attribute are drawn from a normal distribution having mean $\mu$ and variance $\sigma$. At each time step, the value of the windowed aggregate over the true result ($U$) and over the sampled result ($\hat{U}$) are calculated. The relative error in the aggregate is $|\hat{U} - U|/U$. We report the average of these errors over the entire run. In all experiments, while implementing UNIFORM, we assume a tuple size of 32 bytes. The two extra fields required (see Figure 3.3) are stored compactly in two bytes, thus giving a new tuple size of 34 bytes.

We first consider the case when the aggregated attribute is part of $S_1$. Recall that all the $S_1$-probe join tuples produced by a tuple $s \in S_1$ form a cluster. Thus, in this case, all tuples in a cluster have the same value in the aggregated attribute, which is the worst case for cluster sampling.

**Effect of Allocated Memory**: Figure 3.7 shows the aggregation error of the various sampling approaches as we vary the amount of allocated memory. We use the inversely correlated (IC) frequency-based model with $Z = 2$, and we fix $\mu = \sigma = 100$. We see that PPS-CLUSTER outperforms EQ-CLUSTER: in the IC case, there are a small number of large clusters in the result which may be missed by EQ-CLUSTER. UNIFORM performs better than PPS-CLUSTER when the allocated memory is 10%. However, the fraction that can be sampled grows more rapidly for PPS-CLUSTER than for UNIFORM. Consequently, PPS-CLUSTER performs better at higher allocated memory. Note that the error of UNIFORM does not go down to 0 even when allocated memory is 100%. This is because even the synthetic data does not adhere perfectly to the model, as is required for the correctness of UNIFORM (Section 3.4.3).

**Effect of Population Variance**: Figure 3.8 shows the aggregation error of the various sampling approaches as the variance of the aggregated attribute is varied. We show the variance normalized by the mean, i.e., we show the coefficient of variation ($\sigma/\mu$). The allocated memory is 10%, $\mu = 100$, and the model used is the uncorrelated (UC) frequency-based model with $Z = 2$. As the population variance increases, since all tuples in a cluster have the same value, the inter-cluster variance increases. As a result, the performance of cluster sampling approaches degrades as compared to UNIFORM.

Finally, note that for comparing our sampling approaches, we have calculated the exact aggregate over the sampled result. In reality, when memory is limited, this aggregation
CHAPTER 3. MEMORY-LIMITED EXECUTION OF WINDOWED STREAM JOINS

Figure 3.7: Aggregation error vs. memory allocated, $IC$ frequency-based model, $Z = 2$, $\mu = \sigma = 100$

may be approximated [46].

3.6.2.5 Memory Allocation across Multiple Joins

For memory allocation among multiple joins, we study the performance of our ALLOC scheme in comparison with simple proportional memory allocation (PROP). We only study the case when the approximation metric of each join is the recall obtained in that join.

**Frequency-Based Model:** We allocate memory across two joins that follow the frequency-based model: one follows the directly correlated (DC) case, and the other, the inversely correlated (IC) case (recall Section 3.6.2.3). The total available memory is 20% of that required for executing both joins accurately. The load-shedding strategy used for each join is PROB [45]. Figure 3.9 shows a comparison of the minimum recall obtained by both approaches when we vary the skew parameter ($Z$) of the frequency-based model. As $Z$ increases, the minimum recall remains almost constant for ALLOC, but decreases sharply for PROP. The amount of memory allocated to each join by ALLOC (as a percentage of the total memory required) is shown by the dashed plots on the secondary Y-axis. Note that PROP always splits the available memory evenly between the two joins, i.e., 10% to each join.

To understand these results, notice that the IC case is “easy”, i.e., a relatively higher
3.6. EXPERIMENTS

Figure 3.8: Aggregation error vs. population variance, UC frequency-based model, $Z = 2$, $\mu = 100$, Memory=10%

recall can be produced using a small amount of memory: only the rare values of $S_1$ (which are frequent on $S_2$) need to be retained. In contrast, the DC case is “hard”, i.e., more memory is required to obtain the same recall because the common values on $S_1$ need to be retained. Moreover, as the skew ($Z$) increases, the IC case becomes easier, and the DC case becomes harder. ALLOC is able to outperform PROP by allocating less memory to the IC case, and using this extra memory to boost the performance of the DC case.

**Age-Based Model:** We allocate memory across two joins that follow the age-based model, one with an increasing (INC) age curve, and another with a decreasing (DEC) one. The INC curve is chosen as $p_1(k) \propto k^p$ and the DEC curve as $p_1(k) \propto (m - k)^p$, where the exponent $p$ is varied. The total available memory is 50% of that required for executing both joins accurately. The load-shedding strategy used for each join is AGE (Section 3.3.2.1). Figure 3.10 shows a comparison of the minimum recall obtained by both approaches when we vary the exponent $p$. As $p$ increases, the minimum recall increases for ALLOC but remains constant for PROP. With increase in $p$, the DEC case becomes “easier”, while the INC case remains equally “hard” (by Equation 3.3). Thus ALLOC is able to outperform PROP by allocating less memory to DEC, and using the extra memory to boost the performance of INC.

**More Joins:** We omit the results of experimenting with a greater number of joins, as the
findings were similar: As more “hard” joins are added, the gain of \textit{ALLOC} over \textit{PROP} decreases, while if more “easy” joins are added, the gain of \textit{ALLOC} over \textit{PROP} increases. Intuitively, the performance of \textit{PROP} is always limited by the hardest join, while \textit{ALLOC} equalizes the recall among all joins.

3.7 Related Work

The body of work related to answering queries approximately when available memory is insufficient can be broadly classified into two categories. One category consists of load-shedding strategies for max-subset approximation of joins. Random load shedding is the simplest, and has been considered in [81]. [45] primarily considers the offline load-shedding problem (one in which all future tuple arrivals are known), and provides some heuristics for the online case that implicitly assume a frequency-based model. An alternative stream model for load shedding uses a stochastic process [128]. Although this model is more general, the primary focus in [128] is on scenarios in which the tuples arriving on one stream are independent of those that have already arrived on another stream. Most scenarios we consider do not exhibit this independence, e.g., our age-based example in Section 3.2.2. Moreover, the process of inferring a general stochastic process merely by observing the stream is difficult—we would first need to guess a specific stochastic process,
3.7. RELATED WORK

and then infer the parameters of that process by observing the stream. In contrast, the frequency-based and age-based models are far simpler and their parameters can easily be inferred by profiling (Section 3.2.3).

The other category consists of randomized sketch-based solutions for approximately answering aggregate queries over joins, providing probabilistic error guarantees [7, 53]. These techniques do not extend to handle sliding-window joins or windowed aggregates, which are required in many applications: although the techniques handle explicit deletions within streams, they cannot handle the implicit deletions generated by sliding windows. We take a different route: we develop techniques to extract a random sample of the sliding-window join result, and use this sample to get an estimate of the aggregate. Our techniques may also be used when the join result is not being aggregated, but the application requires a statistically random sample of the join result.

In this chapter, we only consider the case of the DSMS being memory-limited. The DSMS could instead (or also) be CPU-limited, i.e., the rate of incoming tuples is higher than can be processed. Load-shedding for the CPU-limited case has been considered in [20, 118]. Sampling from a window is addressed in [19], but only for a single stream and not for a join result. Approximately answering aggregate queries over a single stream is addressed through exponential histograms in [46]. Random sampling for joins has been considered in the relational context [34]. However, all sampling methods developed there

Figure 3.10: Memory allocation across joins: age-based model, Memory=50%
require repeated access or indices on at least one of the relations, making the techniques inapplicable in the context of data streams.

3.8 Conclusion

In this chapter, we addressed the problem of computing approximate answers to sliding-window join queries when the available memory is insufficient to retain the entire state required by the join. We outlined two interesting types of approximation: obtaining a maximum-size subset of the join result, or obtaining a maximum-size random sample of the join result. We showed formally that neither of these approximation goals can be met effectively when the stream arrivals are arbitrary. We defined a novel age-based model that often enables us to address the max-subset problem more effectively than the frequency-based model used previously. We addressed the random-sampling problem for both the age-based as well as the frequency-based models. We also gave an algorithm for memory allocation across joins to minimize the maximum approximation error. Finally, we gave a thorough experimental evaluation of our techniques where we demonstrated that on both real and synthetic data, our algorithms can obtain a larger subset of the join result (or lower errors in aggregation for the case of sampling) than previously known or naïve approaches. Our experiments also demonstrated that our scheme for memory allocation across joins gives significantly better overall accuracy than simple proportional allocation of memory.
Chapter 4

Operator Placement for In-Network Query Processing

So far in the thesis, we have considered a setting in which all the data to be queried is streamed to a central server where the DSMS is running and continuous queries are registered. In this chapter, we move to a distributed setting such as a monitoring application in which data is acquired at multiple nodes that may be constrained in computational and communication power. In such a setting, due to power constraints and the high cost of communication, it may not be desirable to stream all the data through the network to a central server and analyze it there. Instead, it may be more efficient to perform in-network query processing, i.e., to execute query operators on the data en-route to its final destination. In-network query processing balances the benefits of reduced data transmission cost by performing operators in the network against the generally higher cost of processing at low-capability devices. In this chapter, for a class of queries involving filters and joins, we consider the problem of operator placement for in-network query processing, i.e., deciding where each query operator should be executed such that the overall resource consumption is minimized.

4.1 Introduction

In real-world monitoring applications, data acquisition frequently takes place at “edge devices” with limited capabilities, such as sensors, RFID readers, or cameras. Collected data is transmitted through a hierarchy of network nodes and links with progressively
increasing computational power and network bandwidth, as shown in Figure 4.1. The “high fan-in” environment addressed by the Berkeley HiFi project [61] captures this type of application. Other scenarios that involve data acquisition and subsequent processing, e.g., network monitoring [43], can exhibit similar general characteristics.

Typically, queries are posed and results collected at the root of the hierarchy. One simple approach to query execution is to transmit all acquired data through the hierarchy to the root, then perform all processing at the root. However, if queries produce significantly less data than they consume—because of filtering, aggregation, or low-selectivity joins—then this approach may pose considerable unnecessary burden on network bandwidth. In-network query processing pushes some or all of the query execution task to nodes lower in the hierarchy, in order to reduce overall network costs [88]. In general, finding the best point in the hierarchy at which to perform specific processing is a difficult problem: Placing work lower in the hierarchy reduces transmission costs but imposes more burden on lower-capability devices. The goal is to properly balance these opposing effects to minimize overall cost.

Previous work on in-network query processing has focused on queries in which data reduction occurs because of aggregation operators [88], or inexpensive filters. Since CPU costs are typically much lower than communication costs, it is common in such cases to perform all operations as low down in the hierarchy as possible. For example, a common heuristic is to push down all filters to the leaf nodes or sensors. However, for queries that may involve expensive filters, such as text, image, or video filtering, or lookups to remote sources, it may not be best (or even possible) to filter at the low-capability leaf nodes or sensors.
4.1. INTRODUCTION

As a concrete example, consider a video surveillance application running over a hierarchy of nodes, with the sensors capturing the raw images. Suppose the user is looking for images in which the monitored area is dimly lit and there is a lot of motion between successive frames (indicating potentially suspicious activity). This query involves two filters. The first and relatively cheap filter (say $F_1$) checks for dim images by calculating the average pixel intensity. The second filter (say $F_2$) checks for “sufficient” motion, and may be a complicated image processing algorithm that is best not run on the low-capability image-gathering sensor. In this case, the preferred method of execution would be to execute $F_1$ at the sensor, or as close to it as possible, and then transmit filtered images up the hierarchy to a node with sufficient computational power to execute $F_2$.

In general, our objective is to place each query operator at the “best” node in the hierarchy for that operator, based on its selectivity and cost, so that the total cost of computation and communication is minimized. We refer to this problem as the operator placement problem. Intuitively, the main tradeoff that needs to be considered is the lower computational costs at nodes higher up in the hierarchy against the transmission cost of getting the data up to those nodes. Suppose we have an $m$-level hierarchy and $n$ operators in our query. Then there are $m^n$ possible filter placements for processing the data as it travels to the root. We show that nevertheless the operator placement problem has a polynomial-time optimal solution. We provide an algorithm that finds the optimal placement, and show that a simpler greedy algorithm can fail to find the optimal solution.

A key idea in our work is to model network links as filters. Then we can address our overall problem as one of filter ordering on a single node, but with precedence constraints for those filters that are modeling links. We start by considering queries that are conjunctions of uncorrelated filters, i.e., filters whose selectivity is independent of the other filters. We show how the precedence constraints can be dealt with so that known results on filter ordering [76] can be reused to yield the optimal operator placement. We then consider correlated filters. When filters are correlated, even the simpler problem of optimal ordering of correlated filters at a single node is NP-hard [57] and can at best be approximated within a factor of 4 [98]. Hence, we cannot hope to find the optimal solution to our distributed version of the problem in polynomial time. However, we show that our technique of modeling network links as filters extends to the correlated-filters case and yields the best-possible 4-approximation to the optimal operator placement. Finally, we extend our algorithm to handle queries that include multiway joins in addition to filters.
how to decide where a join operator should be placed optimally with respect to the query’s filter operators.

Chapter Organization

The rest of the chapter is organized as follows. In Section 4.2, we formally define the problem of operator placement for in-network processing of queries with expensive filters. In Section 4.3, we first describe a greedy algorithm that can fail to find the globally optimal solution to the operator placement problem, and then present our polynomial-time optimal algorithm that handles queries with uncorrelated filters. We also extend our algorithm to provide the best possible approximation for correlated filters. In Section 4.4, we extend our algorithm to handle queries that include a multiway stream join together with filters. In Section 4.5, we outline several practical, interesting variations of the basic operator-placement problem, and prove the hardness of solving these variants, thereby showing that this area is ripe for future research. The variants we consider include nodes with resource constraints, load balancing across nodes, and a more complex cost model for how filter costs may vary across different nodes. We finally present related work in Section 4.6 and conclude in Section 4.7.

4.2 Preliminaries

We begin by considering data acquired by only one of the leaf nodes of Figure 4.1 and focus on in-network query processing over this data. As this data is transmitted up the hierarchy, the basic network topology we need to consider (shown in Figure 4.2) consists of a linear chain of nodes \( N_1, N_2, \ldots, N_m \), where \( m \) is the number of levels in the hierarchy.
4.2. PRELIMINARIES

In relation to Figure 4.1, the leftmost node $N_1$ corresponds to the point of acquisition, while the rightmost node $N_m$ corresponds to the root of the hierarchy. Each node $N_j$ transmits only to node $N_{j+1}$. We consider the linear hierarchy merely for ease of presentation; in Section 4.3.4 we show how our algorithms extend in a straightforward manner to general tree hierarchies.

Let $S$ denote the stream of data items acquired by node $N_1$. Let $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ be a set of $n$ filters. We first consider in-network processing for the following basic query posed at the root node $N_m$.

\[
\text{SELECT } * \text{ FROM } S \text{ WHERE } F_1 \land F_2 \land \ldots \land F_n
\] (4.1)

In Section 4.4 we extend our algorithms to deal with queries that involve a multiway join of streams in addition to conjunctive filters. Unlike Chapter 2, in this chapter we do not consider multiple queries together: The possibility of shared computation among multiple queries yields an even more complex operator placement problem, but it may also yield even higher benefits and thus is a promising direction for future work.

An in-network query plan for the query in (4.1) is simply a mapping of each filter in $\mathcal{F}$ to exactly one node. Figure 4.3 shows a sample in-network query plan for executing a query with $n = 4$ filters on $m = 4$ nodes. Figure 4.3 also shows the data that is transmitted along each network link. Each link transmits only those tuples that have passed all filters executed so far. The cost of an in-network query plan consists of two parts: the cost of executing the filters on the various nodes, together with the cost of transmitting the tuples over the network links. The exact model used to evaluate the cost of an in-network query plan is explained in the next section.
4.2.1 Cost Model

To derive the cost of an in-network query plan, we assume, as in Chapter 2, that for each filter $F_i \in \mathcal{F}$, the following two quantities are known:

1. **Cost**: The average per-item cost of evaluating filter $F$ on node $N_i$ is denoted by $c(F, i)$. To model the fact that the nodes in the hierarchy have increasing computational power, we assume that the cost of any filter scales down by a factor $\gamma_i < 1$ on moving from node $N_i$ to $N_{i+1}$ (see Figure 4.2). That is, $c(F, i + 1) = \gamma_j c(F, i)$. Note that even though we are supposing scale-down, a decrease in computational power on moving from node $N_i$ to $N_{i+1}$ is captured by $\gamma_i > 1$ and can be incorporated into our approach directly.

2. **Selectivity**: The average fraction of data items on stream $S$ that satisfy filter $F$ is referred to as the selectivity of filter $F$, and is denoted by $s(F)$. We assume for now that the filters are independent, i.e., selectivity of a filter remains the same irrespective of which filters have been applied earlier. Correlated filters are dealt with in Section 4.3.3.

Apart from the above, the cost of a plan also depends on the following quantity:

- **Cost of network transmission**: The cost of transmitting a tuple on the network link from node $N_i$ to $N_{i+1}$ is $l_i$ (see Figure 4.2). We assume that $l_i$ includes an appropriate multiplicative factor to convert transmission cost into a quantity that can be treated at par with computational cost.

Consider an in-network query plan $\mathcal{P}$ for the query in (4.1). We define:

<table>
<thead>
<tr>
<th>$\mathcal{P}(F)$</th>
<th>Index number of the node at which filter $F$ is executed under plan $\mathcal{P}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{F}_i$</td>
<td>Set of filters executed at node $N_i$, i.e., $\mathcal{F}_i = {F \in \mathcal{F}</td>
</tr>
</tbody>
</table>

We assume that at each node $N_i$, the set of filters $\mathcal{F}_i$ are executed in the optimal sequence given by the following theorem [76].

**Theorem 4.2.1.** The optimal sequence to execute a set of independent filters on a single node is in increasing order of rank, where rank of a filter $F$ is given by $\text{rank}(F) = \frac{\text{cost}(F)}{1-\text{selectivity}(F)}$. □
Consider a sequence of \( n' \) filters \( F' = F'_1, \ldots, F'_n \). Let \( c(F', i) \) denote the cost per tuple of executing this sequence at node \( N_i \). It is given by:

\[
c(F', i) = \sum_{j=1}^{n'} \left( \prod_{k=1}^{j-1} s(F'_k) \right) c(F'_j, i)
\]  

(4.2)

Let \( r(F_i) \) denote the sequence of filters in \( F_i \) in rank order. Without loss of generality assume that the data in stream \( S \) is acquired at the rate of one tuple per some unit time. Then the cost per unit time of the in-network plan \( P \) is given by (assume \( l_0 = 0 \)):

\[
c(P) = \sum_{i=1}^{m} \left( \prod_{F \in F: P(F) < i} s(F) \right) \left( l_{i-1} + c(r(F_i), i) \right)
\]  

(4.3)

**Example 4.2.2.** Consider the in-network query plan shown in Figure 4.3. Let the selectivity of each filter be \( 1/2 \), and let the costs at node \( N_1 \) of the filters be:

<table>
<thead>
<tr>
<th>( F )</th>
<th>( F_1 )</th>
<th>( F_2 )</th>
<th>( F_3 )</th>
<th>( F_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c(F, 1) )</td>
<td>200</td>
<td>400</td>
<td>1300</td>
<td>2500</td>
</tr>
</tbody>
</table>

The cost scaling factors and the transmission costs are as shown in Figure 4.3. Assume stream tuples are acquired at \( N_1 \) at unit rate.

Using equation (4.2), the execution cost of the sequence \( F_1, F_2 \) of filters at node \( N_1 \) is \( 200 + \frac{1}{2} \cdot 400 = 400 \). Since two filters each with selectivity \( 1/2 \) have been applied, the rate of data transmitted from \( N_1 \) to \( N_2 \) and from \( N_2 \) to \( N_3 \) is \( 1/4 \) of the unit rate each. Thus the total transmission cost up to node \( N_3 \) is \( \frac{1}{4} (700 + 500) = 300 \). The per-tuple execution cost of \( F_3 \) at \( N_3 \) is \( c(F_3, 3) = \gamma_1 \gamma_2 c(F_3, 1) = 130 \). Since the rate into \( N_3 \) is \( 1/4 \), the execution cost of \( F_3 \) is \( \frac{1}{4} \cdot 130 = 32.5 \). Similarly the transmission cost from \( N_3 \) to \( N_4 \) and the execution cost of \( F_4 \) are calculated to be 37.5 and 7.8 respectively. Thus the total cost is \( c(P) = 400 + 300 + 32.5 + 37.5 + 7.8 = 777.8 \).

**4.2.2 Problem Statement**

Since each of the \( n \) filters can be placed at any of the \( m \) nodes, there are \( m^n \) possible in-network query plans. The problem of operator placement for in-network query processing

---

\(^1\)Since the filters have different costs at different nodes, the actual rank of a filter is node-dependent. However, since the cost of each filter scales by the same factor going from one node to the next, the rank order of filters remains the same at every node. In Section 4.5.2 we discuss a more general model in which each filter’s cost may scale differently across nodes.
is to efficiently choose the least-cost plan among the exponential number of alternatives.

**Definition 4.2.3 (Operator Placement Problem).** For each filter \( F \in \mathcal{F} \), choose \( P(F) \in \{1, \ldots, m\} \) such that \( c(P) \) given by (4.3) is minimized.

### 4.3 Filter Placement

In this section, we consider solutions to the operator placement problem given by Definition 4.2.3. We first assume independent filters and specify a local greedy operator placement algorithm (Section 4.3.1). We show that this algorithm does not always find the globally optimal solution. We then provide an optimal operator placement algorithm (Section 4.3.2), and extend this algorithm for correlated filters (Section 4.3.3) and tree hierarchies (Section 4.3.4).

#### 4.3.1 Greedy Algorithm

For an in-network query plan \( \mathcal{P} \), let \( c(\mathcal{P}, i) \) denote the part of the total cost \( c(\mathcal{P}) \) that is incurred at node \( N_i \). This cost includes not only the execution of filters at node \( N_i \), but also the transmission of the filtered tuple stream from node \( N_i \) to \( N_{i+1} \). \( c(\mathcal{P}) = \sum_{i=1}^{m} c(\mathcal{P}, i) \), and notice that \( c(\mathcal{P}, i) \) depends only on \( \mathcal{F}_1, \ldots, \mathcal{F}_i \) and not on \( \mathcal{F}_{i+1}, \ldots, \mathcal{F}_m \).

A simple but reasonable way to approach the operator placement problem is the following greedy algorithm. Start with node \( N_1 \) and choose a set of filters \( \mathcal{F}_1 \) so that \( c(\mathcal{P}, 1) \) is minimized (explained in the next paragraph). Then apply the approach recursively with nodes \( \{N_2, \ldots, N_m\} \) and the set of filters \( \mathcal{F} - \mathcal{F}_1 \). Our global objective is to minimize \( \sum_{i=1}^{m} c(\mathcal{P}, i) \); the greedy algorithm minimizes each \( c(\mathcal{P}, i) \) individually in increasing order of \( i \). In other words, the greedy algorithm decides which filters to apply by balancing filtering cost against the cost of transmitting unfiltered data to the next node, but it does not take into account how much cheaper it would be to filter the data further up the hierarchy.

For minimizing \( c(\mathcal{P}, 1) \) in the base case of the recursion, we introduce a key idea behind all our algorithms: modeling network links as filters. Logically, we construct a filter corresponding to each network link, such that transmitting a tuple over the link is equivalent in terms of cost to executing the constructed filter over the tuple. For cost evaluation, the entire in-network query plan can then be treated as executing a sequence of filters on a single node, enabling us to leverage previous work on filter ordering [36, 76, 98].
4.3. FILTER PLACEMENT

To minimize \( c(\mathcal{P}, 1) \), we model the network link from node \( N_1 \) to \( N_2 \) as a filter \( F^l_1 \) with cost \( c(F^l_1, 1) = l_1 \) (as the cost of transmitting a tuple over the link is \( l_1 \)), and selectivity \( s(F^l_1) = 0 \). Choosing a selectivity of 0 for \( F^l_1 \) factors away the cost of the plan processing at nodes \( N_2, \ldots, N_m \), and thus enables the greedy algorithm to optimize only for \( c(\mathcal{P}, 1) \) separately. We now show that \( c(\mathcal{P}, 1) \) can be written as the cost of executing the filters in \( \mathcal{F}_1 \) followed by the filter \( F^l_1 \) at node \( N_1 \).

**Lemma 4.3.1.** Construct \( F^l_1 \) with \( s(F^l_1) = 0 \), \( c(F^l_1, 1) = l_1 \). Then \( c(\mathcal{P}, 1) = c(r(\mathcal{F}_1) \bullet F^l_1, 1) \) where \( \bullet \) denotes concatenation of sequences.

**Proof.** From (4.3),

\[
c(\mathcal{P}, 1) = c(r(\mathcal{F}_1), 1) + \prod_{F \in \mathcal{F} \setminus \mathcal{P}(F) < 2} s(F) l_1
\]

\[
= c(r(\mathcal{F}_1), 1) + \prod_{F \in \mathcal{F}_1} s(F) c(F^l_1, 1)
\]

\[
= c(r(\mathcal{F}_1) \bullet F^l_1, 1)
\]

We then order \( F^l_1 \) and the filters in \( \mathcal{F} \) based on rank (recall Theorem 4.2.1) and choose as \( \mathcal{F}_1 \) all the filters that occur before \( F^l_1 \) in rank order. Note that since \( \text{rank}(F^l_1) = l_1 \), effectively we simply choose as \( \mathcal{F}_1 \) all filters that have rank \( < l_1 \).

**Theorem 4.3.2.** \( c(\mathcal{P}, 1) \) is minimized when:

\[
\mathcal{F}_1 = \{ F \mid F \text{ occurs before } F^l_1 \text{ in } r(\mathcal{F} \cup \{ F^l_1 \}) \}
\]

**Proof.** Consider a plan \( \mathcal{P} \) in which \( \mathcal{F}_1 \) is chosen according to the theorem statement. By Lemma 4.3.1, \( c(\mathcal{P}, 1) = c(r(\mathcal{F}_1) \bullet F^l_1, 1) \). Since \( s(F^l_1) = 0 \), we can append any number of filters after \( F^l_1 \) without changing the cost of executing the sequence. Thus we can write:

\[
c(\mathcal{P}, 1) = c(r(\mathcal{F}_1) \bullet F^l_1 \bullet r(\mathcal{F} - \mathcal{F}_1), 1) \quad (4.4)
\]

Now suppose for contradiction that there is a different set of filters \( \mathcal{F}'_1 \) to be executed at node \( N_1 \) and a corresponding in-network query plan \( \mathcal{P}' \) such that \( c(\mathcal{P}', 1) < c(\mathcal{P}, 1) \). Similar to (4.4), we can write:

\[
c(\mathcal{P}', 1) = c(r(\mathcal{F}'_1) \bullet F^l_1 \bullet r(\mathcal{F} - \mathcal{F}'_1), 1) \quad (4.5)
\]
CHAPTER 4. OPERATOR PLACEMENT FOR IN-NETWORK QUERY PROCESSING

Procedure **RECURSIVE**($\mathcal{F}, \{N_1, \ldots, N_m\}$)
1. if ($m = 1$) $\mathcal{F}_m = \mathcal{F}$; return
2. Construct $\mathcal{F}_1^l$ with $s(\mathcal{F}_1^l) = 0$ and $c(\mathcal{F}_1^l, 1) = l_1$
3. Calculate $r(\mathcal{F} \cup \{\mathcal{F}_1^l\})$ for ranks at node $N_1$
4. $\mathcal{F}_1 = \{F \mid F$ occurs before $\mathcal{F}_1^l$ in $r(\mathcal{F} \cup \{\mathcal{F}_1^l\})\}$
5. **RECURSIVE**($\mathcal{F} - \mathcal{F}_1, \{N_2, \ldots, N_m\}$)

Algorithm **GREEDY**
1. **RECURSIVE**($\mathcal{F}, \{N_1, \ldots, N_m\}$)

Figure 4.4: Greedy Algorithm: $O((m + n) \log(n))$

The right sides of (4.4) and (4.5) give the execution cost of the same set of filters $\mathcal{F} \cup \{\mathcal{F}_1^l\}$ but in different sequences. By the choice of $\mathcal{F}_1$, the sequence in (4.4) is rank ordered, but that in (4.5) is not. By Theorem 4.2.1, $c(\mathcal{P}, 1) \leq c(\mathcal{P}', 1)$. Thus we get a contradiction.

A summary of the greedy algorithm is shown in Figure 4.4. We illustrate the operation of the greedy algorithm by an example.

**Example 4.3.3.** Consider operator placement using the greedy algorithm for Example 4.2.2. The ranks of $F_1, \ldots, F_4$ at $N_1$ are 400, 800, 2600, and 5000 respectively. The rank of $F_1^l$ is $l_1 = 700$. Thus $\mathcal{F}_1$ is chosen as $\{F_1\}$. The ranks of $F_2, \ldots, F_4$ at $N_2$ are obtained by scaling down the ranks at $N_1$ by $\gamma_1$, so they are 160, 520, and 1000. Only rank($F_2$) < $l_2$, thus $\mathcal{F}_2 = \{F_2\}$. Continuing in this fashion, we obtain $\mathcal{F}_3 = \{F_3\}$ and $\mathcal{F}_4 = \{F_4\}$.

For this plan, we find $c(\mathcal{P}) = 792.8$ by (4.3).

The greedy algorithm makes very local decisions. Thus it is not surprising that the greedy algorithm does not always produce the globally optimal solution. For instance, $c(\mathcal{P}) = 792.8$ in Example 4.3.3 is greater than $c(\mathcal{P}) = 777.8$ in Example 4.2.2.

### 4.3.2 Optimal Algorithm

In the greedy algorithm of Section 4.3.1, network links are modeled as filters with selectivity 0. This approach enables us to capture the transmission cost of the link, but the remainder of the tuple processing cost (at nodes further up in the hierarchy) is not captured. Thus we can only get an expression for $c(\mathcal{P}, 1)$ in terms of the execution cost of a
sequence of filters (Lemma 4.3.1), but not an expression for the entire $c(P)$. The optimal algorithm we present relies on obtaining an analogous expression for $c(P)$.

Assume $\gamma_i \leq 1$ for each $i$ ($\gamma_i > 1$ is handled in Section 4.3.2.2). Since $c(F, i + 1) = \gamma_i c(F, i)$, transmitting data on the link from node $N_i$ to $N_{i+1}$ cuts down by a factor $\gamma_i$ the per-tuple cost of any filter applied subsequently. In terms of cost per unit time, this cost scale-down is equivalent to the stream rate slowing down by a factor $\gamma_i$, but the filter costs themselves remaining unchanged. Hence the link from node $N_i$ to $N_{i+1}$ can be modeled as a filter $F_i^l$ with $s(F_i^l) = \gamma_i$. Additionally, we set $c(F_i^l, 1) = l_i (\prod_{j=1}^{i-1} \gamma_j)^{-1}$. Intuitively, the per-tuple cost of traversing the link is $l_i$, even after the previous network links have been traversed. Thus the term $(\prod_{j=1}^{i-1} \gamma_j)^{-1}$ is present to compensate for the scale-down produced by the filters $F_1^l, \ldots, F_{i-1}^l$. We can now write $c(P)$ in terms of the execution cost of a sequence of filters (assume all ranks are calculated at $N_1$).

**Lemma 4.3.4.** For $i \in \{1, \ldots, m - 1\}$ construct $F_i^l$ with $s(F_i^l) = \gamma_i$, $c(F_i^l, 1) = l_i (\prod_{j=1}^{i-1} \gamma_j)^{-1}$. Then:

$$c(P) = c(r(F_1) \cdot F_1^l \cdot r(F_2) \cdot \ldots \cdot F_{m-1}^l \cdot r(F_m), 1)$$

where $\cdot$ denotes concatenation of sequences.

**Proof.** Since $c(F, i + 1) = \gamma_i c(F, i)$ for any $F \in \mathcal{F}$, we have $c(r(F), i) = \prod_{j=1}^{i-1} \gamma_j c(r(F), 1)$. From (4.3), $c(P)$ is given by (assume $\gamma_0 = 1$):

$$\sum_{i=1}^{m} \prod_{F \mid |P(F) < i} s(F) \left( l_{i-1} + \prod_{j=1}^{i-1} \gamma_j c(r(F), 1) \right)$$

$$= \sum_{i=1}^{m} \prod_{F \mid |P(F) < i} s(F) \prod_{j=1}^{i-2} \gamma_j \left( c(F_{i-1}^l, 1) + \gamma_{i-1} c(r(F), 1) \right)$$

$$= \sum_{i=1}^{m} \prod_{F \mid |P(F) < i} s(F) \prod_{j=1}^{i-2} s(F_j^l) \left( c(F_{i-1}^l, 1) + s(F_{i-1}^l) c(r(F), 1) \right)$$

$$= c(r(F_1) \cdot F_1^l \cdot r(F_2) \cdot \ldots \cdot F_{m-1}^l \cdot r(F_m), 1) \quad \square$$

Suppose for now that the ranks of the sequence of filters $F_1^l, \ldots, F_{m-1}^l$ (modeling links)
are in non-decreasing order. Then the following result and its proof are analogous to Theorem 4.3.2.

**Theorem 4.3.5.** Suppose \( \text{rank}(F_i^l) < \text{rank}(F_{i+1}^l) \) for each \( i \in \{1, \ldots, m-2\} \). Denote by \( \mathcal{F}' \) the filter sequence \( F_0^l \bullet r(\mathcal{F} \cup \{F_1^l, \ldots, F_{m-1}^l\}) \bullet F_m^l \). Then \( c(\mathcal{P}) \) is minimized when:

\[
F_i = \{ F \mid F \text{ occurs between } F_{i-1}^l \text{ and } F_i^l \text{ in } \mathcal{F}' \}
\]

**Proof.** Consider a plan \( \mathcal{P} \) in which each \( \mathcal{F}_i \) is chosen according to the theorem statement. By Lemma 4.3.4,

\[
c(\mathcal{P}) = c(r(\mathcal{F}_1) \bullet F_1^l \bullet r(\mathcal{F}_2) \bullet \ldots \bullet F_{m-1}^l \bullet r(\mathcal{F}_m), 1) \tag{4.6}
\]

Now suppose for contradiction that there is a different assignment of filters \( \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m \) to the nodes and a corresponding in-network query plan \( \mathcal{P}' \) such that \( c(\mathcal{P}', 1) < c(\mathcal{P}, 1) \). Similar to (4.6), we can write:

\[
c(\mathcal{P}', 1) = c(r(\mathcal{F}_1') \bullet F_1^l \bullet r(\mathcal{F}_2') \bullet \ldots \bullet F_{m-1}^l \bullet r(\mathcal{F}_m'), 1) \tag{4.7}
\]

The right sides of (4.6) and (4.7) give the execution cost of the same set of filters \( \cup_{i=1}^{m-1} F_i^l \cup \mathcal{F} \) but in different sequences. By the choice of \( \mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m \), the sequence in (4.6) is rank ordered, but that in (4.7) is not. By Theorem 4.2.1, \( c(\mathcal{P}, 1) \leq c(\mathcal{P}', 1) \). Thus we get a contradiction.

In general the ranks of \( F_1^l, \ldots, F_{m-1}^l \) may not be in non-decreasing order. To deal with such cases, we introduce the concept of “short-circuiting”.

### 4.3.2.1 Short-Circuiting

Suppose \( \text{rank}(F_{i-1}^l) > \text{rank}(F_i^l) \) for some \( i \). We show that in the optimal in-network query plan in this scenario, no filter is executed at node \( N_i \).

**Lemma 4.3.6.** If \( \text{rank}(F_{i-1}^l) > \text{rank}(F_i^l) \) for some \( i \in \{2, \ldots, m-1\} \), then in the optimal plan \( \mathcal{F}_i = \emptyset \).

**Proof.** Suppose \( \text{rank}(F_{i-1}^l) > \text{rank}(F_i^l) \) and in the optimal in-network query plan \( \mathcal{P}, \mathcal{F}_i \neq \emptyset \). Consider the alternate query plans \( \mathcal{P}' \) and \( \mathcal{P}'' \) which are the same as \( \mathcal{P} \) except that the filters...
4.3. FILTER PLACEMENT

in $\mathcal{F}_i$ have been moved to node $N_{i-1}$ and $N_i$ respectively. We have

$$c(\mathcal{P}) = a_1 \left( l_{i-1} + c(r(\mathcal{F}_i), i) + a_2 l_i \right) + a_3$$

where $a_1 = \prod_{F \mid P \subset \mathcal{F}_i} s(F), a_2 = \prod_{F \mid F \subset N_i} s(F),$ and $a_3$ denotes the sum of the other terms in $c(\mathcal{P})$ from (4.3). Similarly:

$$c(\mathcal{P}') = a_1 \left( c(r(\mathcal{F}_i), i) \gamma_{i-1}^{-1} + a_2 (l_{i-1} + l_i) \right) + a_3$$

$$c(\mathcal{P}'') = a_1 \left( l_{i-1} + l_i + c(r(\mathcal{F}_i), i) \gamma_i \right) + a_3$$

Since $\mathcal{P}$ is optimal, we must have $c(\mathcal{P}) < c(\mathcal{P}')$ and $c(\mathcal{P}) < c(\mathcal{P}'').$ Substituting for $c(\mathcal{P}), c(\mathcal{P}'),$ and $c(\mathcal{P}'')$ and simplifying, we get:

$$\frac{l_{i-1} \gamma_{i-1}}{1 - \gamma_{i-1}} < \frac{l_i}{1 - \gamma_i}$$  \hfill (4.8)

(4.8) implies that $\text{rank}(F_{i-1}^l) < \text{rank}(F_i^l),$ a contradiction.  

If $\mathcal{F}_i$ is guaranteed to be empty in the optimal query plan, we can modify the network topology by “short-circuiting” node $N_i$ as shown in Figure 4.5. Logically, node $N_i$ is removed, $N_{i-1}$ is connected to node $N_{i+1}$ by a link having cost $l_{i-1} + l_i,$ and the cost scale-down factor from node $N_{i-1}$ to $N_{i+1}$ is set to $\gamma_{i-1} \gamma_i.$ At each short-circuit the number of nodes $m$ decreases by 1.

We can continue short-circuiting on the modified topology until there does not exist any $i$ for which $\text{rank}(F_{i-1}^l) > \text{rank}(F_i^l).$ At that point, Theorem 4.3.5 can be applied to yield the optimal solution.
Algorithm *OPT_FILTER*
1. while (∃ i | γ_i > 1)
2. short-circuit node N_{i+1}
3. while (true)
4. for i = 1 to m - 1
5. s(F_i) = γ_i and c(F_i, 1) = l_i(Π_{j=1}^{i-1} γ_j)^{-1}
6. if (∃ i | rank(F_{i-1}) > rank(F_i))
7. short-circuit node N_i
8. else break
9. F' = F_0 • r(F ∪ {F_1, ..., F_{m-1}}) • F_m
10. for i = 1 to m
11. F_i = { F | F occurs between F_{i-1} and F_i in F' }  

Figure 4.6: Optimal Operator Placement Algorithm

### 4.3.2.2 Handling Cost Scaleup

So far we have assumed γ_i ≤ 1 for each i. If γ_i > 1, it is easy to see that in the optimal solution F_{i+1} = ∅, as follows. If any filters are executed at node N_{i+1} they can be moved to node N_i. The new plan will reduce the computational cost (since c(F, i) < c(F, i + 1)) as well as the transmission cost (since more filters are applied earlier reducing the amount of data transmitted). Thus, just as in Section 4.3.2.1, if γ_i > 1, we can short-circuit node N_{i+1} (if γ_{m-1} > 1 we can simply delete node N_m). We can continue short-circuiting until γ_i ≤ 1 for each i.

### 4.3.2.3 Summary and Example

A summary of the entire algorithm is given in Figure 4.6. Its running time is O((m + n) log(m + n)) due to the sorting of filters in rank order in line 9.

**Example 4.3.7.** Continue with Example 4.2.2. We first construct a filter for each network link (line 5):

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(F_i, 1)</td>
<td>700</td>
<td>2500</td>
<td>3000</td>
</tr>
<tr>
<td>s(F_i)</td>
<td>1/5</td>
<td>1/2</td>
<td>1/4</td>
</tr>
<tr>
<td>rank(F_i)</td>
<td>875</td>
<td>5000</td>
<td>4000</td>
</tr>
</tbody>
</table>
We find that \( \text{rank}(F_2) > \text{rank}(F_3) \). Thus, we can short-circuit \( N_3 \) (line 7). On short-circuiting, we obtain a new link with transmission cost 800 and scale-down factor 1/8 (Figure 4.7). The filter corresponding to this link (denote it by \( F_{2,4} \)) has cost 4000, selectivity 1/8 and hence rank 4571.4. Since \( \text{rank}(F_2) < \text{rank}(F_{2,4}) \), no more short-circuiting is required. The ranks of \( F_1, \ldots, F_4 \) are 400, 800, 2600, and 5000. Thus the rank order of filters is \( F_1, F_2, F_{1,3}, F_3, F_{2,4}, F_4 \) (line 9). Hence \( \mathcal{F}_1 = \{F_1, F_2\}, \mathcal{F}_2 = \{F_3\}, \text{and } \mathcal{F}_3 = \{F_4\} \) (line 11). Since \( N_3 \) has been short-circuited, \( \mathcal{F}_3 = \emptyset \). For this plan, \( c(P) = 747.8 \), that is lower than the costs in Examples 4.2.2 and 4.3.3, and can be verified to be optimal.

4.3.3 Correlated Filters

We now consider operator placement when the filters in \( \mathcal{F} \) may be correlated, i.e., the selectivity of a filter on a stream may depend on the filters that have already been applied. We define the conditional selectivity of a filter \( F \) given a set of filters \( Q \subseteq \mathcal{F} \), denoted \( s(F|Q) \), as the fraction of tuples that satisfy \( F \) given that they satisfy all the filters in \( Q \). Note that if \( F \in Q \), \( s(F|Q) = 1 \).

When filters are correlated, Theorem 4.2.1 no longer holds. In fact, the problem of optimal ordering of correlated filters at a single node has been shown to be NP-hard [57, 98]. The same work also gives a natural greedy algorithm based on conditional selectivity (Figure 4.8) that is guaranteed to find an ordering having a cost at most 4 times the optimal cost. The algorithm defines the conditional rank for each filter (line 3) and at each step, picks the filter having the smallest conditional rank to be executed next. It is also shown that this approximation ratio of 4 is the best possible unless \( P = NP \).

Our problem of optimally executing a set of correlated filters at multiple nodes is clearly at least as difficult as the single-node problem, and hence is NP-hard. We show in this section that the same approximation ratio of 4 can be obtained for our problem setting too. Since algorithm CORRELATE (Figure 4.8) gives the best ordering for correlated
Algorithm \textit{CORRELATED}
\begin{itemize}
  \item $\mathcal{F}$: Set of correlated filters to be ordered
  \item $1. \ Q = \text{set of filters executed so far}$
  \item $2. \ \text{while} (Q \neq \mathcal{F})$
  \item $3. \ \text{conditional rank}(F) = \frac{\text{cost}(F)}{1 - s(F|Q)} \forall F \in \mathcal{F}$
  \item $4. \ F_{\text{min}} = F \in \mathcal{F} \text{ that has smallest conditional rank}$
  \item $5. \ \text{choose } F_{\text{min}} \text{ to be executed next}; \ Q = Q \cup \{F_{\text{min}}\}$
\end{itemize}

Figure 4.8: 4-approximation to Ordering of Correlated Filters

filters that can be found efficiently, we assume that in any in-network query plan, the set of filters at any particular node are executed in the order given by Figure 4.8, i.e., the set of filters at any node are executed in order of conditional rank (conditioned on the set of filters that have already been executed).

We again model the network links as filters with cost and selectivity as before (given by Lemma 4.3.4). Additionally, the filters that model network links are independent of all filters, i.e., for each $i$, $s(F^l_i|Q) = s(F^l_i)$ for any $Q$ such that $F^l_i \notin Q$. Assume as in Section 4.3.2 that all ranks are calculated at node $N_1$. First we show that even in the presence of correlated filters, short-circuiting (Section 4.3.2.1) is still valid.

\textbf{Lemma 4.3.8.} If \( \text{rank}(F^l_{i-1}) > \text{rank}(F^l_i) \) for some $i \in \{2, \ldots, m-1\}$, then in the optimal plan $\mathcal{F}_i = \emptyset$ even if the filters in $\mathcal{F}$ are correlated.

\textit{Proof.} Suppose $\mathcal{F}_i \neq \emptyset$. Replace the filters in $\mathcal{F}_i$ by a single filter $F$ having equivalent per-tuple cost and selectivity as the filter sequence at node $N_i$. Now consider rank$(F)$, which may depend on the filters executed at nodes $N_1, \ldots, N_{i-1}$. Since rank$(F^l_{i-1}) >$ rank$(F^l_i)$, either rank$(F) >$ rank$(F^l_i)$ (in which case move $F$ to $N_{i+1}$) or rank$(F) <$ rank$(F^l_{i-1})$ (in which case move $F$ to $N_{i-1}$). Note that this movement of $F$ does not change the rank of any filter since the ranks of $F^l_{i-1}$ and $F^l_i$ are independent of all filters and the rank of $F$ depends only on the filters executed at nodes $N_1, \ldots, N_{i-1}$ which remain the same. Now using a similar argument as in the proof of Lemma 4.3.6, we see that this movement of $F$ cannot increase the total cost of the solution. Thus $\mathcal{F}_i = \emptyset$ in the optimal solution. \hfill $\square$

So far, for a set of filters $\mathcal{F}'$, we have used $r(\mathcal{F}')$ to denote the sequence of filters in $\mathcal{F}'$ in rank order. For this section, we modify the interpretation of $r(\mathcal{F}')$ to denote the sequence
of filters in $\mathcal{F}'$ in order of conditional rank, i.e., according to algorithm \textsc{Correlated} (Figure 4.8). With this new interpretation of $r(\mathcal{F}')$, we have the following result.

\textbf{Theorem 4.3.9.} Algorithm \textsc{Opt\_Filter} (Figure 4.6) gives a 4-approximation to the optimal operator placement for correlated filters.

\textit{Proof.} With the new interpretation of $r(\mathcal{F}')$ it is easy to see that Lemma 4.3.4 holds in the presence of correlated filters. After short-circuiting, the ranks of the filters $F_i^l, \ldots, F_{m-1}^l$ are non-decreasing. Hence when the filters are ordered (by algorithm \textsc{Correlated}) in line 9 of algorithm \textsc{Opt\_Filter}, the filters corresponding to the network links automatically occur in the desired order, and no ordering restrictions on the filters need to be imposed. Since algorithm \textsc{Correlated} is a 4-approximation to the optimal ordering in the case when there are no ordering restrictions [98], the result follows. \hfill $\square$

\subsection*{4.3.4 Extension to Tree Hierarchies}

So far we have restricted our attention to the data acquired by only one of the leaf nodes or sensors of Figure 4.1. Let $S_i$ denote the stream of data from the $i$th sensor. We have shown how to optimize the query (4.1) over any single stream $S_i$. In reality, query (4.1) may be posed over data gathered by any number of sensors, i.e., the query is $\sigma_\mathcal{F}(S_1 \cup \ldots \cup S_k)$ for $k$ sensors. This query can be written as the union $\sigma_\mathcal{F}(S_1) \cup \ldots \cup \sigma_\mathcal{F}(S_k)$. Each of the queries in this union operates on different data, so there is no opportunity for sharing computation or transmission among these queries. Hence optimizing their combined execution is equivalent to optimizing each of them separately, for which we use the algorithm of Section 4.3.2.

\subsection*{4.4 Joins}

Recall the network topology of Figure 4.2. Now suppose the data acquired by sensor node $N_1$ is in the form of $k$ different data streams (e.g., a temperature stream, a light stream, a vibration stream, and so on). In this section, we consider in-network processing for queries that involve, in addition to filtering, a sliding-window join [18] of these streams. We assume that the join of all $k$ streams is performed at a single node by the \textsc{MJoin} operator [125]; consideration of join trees is left as future work. The sliding-window join operator studied in Chapter 3 is a special case of the \textsc{MJoin} operator when exactly two streams are being
joined. For ease of presentation, we again assume a two-way join; extension to general multiway joins is straightforward.

Let $S_1$ and $S_2$ be the streams acquired by sensor node $N_1$. We consider the query:

$$\text{SELECT } * \text{ FROM } (S_1[W_1] \bowtie S_2[W_2])$$
$$\text{WHERE } F_1 \land \ldots \land F_n$$

(4.9)

where as in Chapter 3, $W_1$ and $W_2$ represent the lengths of the windows (time-based or tuple-based) on streams $S_1$ and $S_2$, and $F = \{F_1, \ldots, F_n\}$ is the set of filters in the query as so far in this chapter. We extend the cost model of Section 4.2.1 to include the selectivity and cost of the join operator.

1. **Selectivity**: The selectivity $s(\bowtie)$ of the join is defined as the fraction of the cross product that occurs in the join result. Thus if streams $S_1$ and $S_2$ have rates of $r_1$ and $r_2$ tuples per unit time coming into the join operator, the output of the join is at rate $s(\bowtie)r_1r_2$.

2. **Cost**: The cost per unit time of performing the join is given by:

$$\text{cost}(\bowtie) = a_1r_1 + a_2r_2 + a_3r_1r_2$$

(4.10)

where $a_1, a_2,$ and $a_3$ are constants. This form arises because a constant amount of work must be done per input tuple (therefore $a_1r_1 + a_2r_2$), and similarly a constant amount of work to output every join tuple (therefore $a_3r_1r_2$). Just as the filter costs, the join cost also scales down by a factor $\gamma_i$ on moving from node $N_i$ to $N_{i+1}$.

Given cost and selectivity for the join operator, we can arrive at an expression, similar to (4.3), that gives the total cost of an in-network query plan for query (4.9). However, the exact expression for the cost is not important for describing our operator-placement algorithm.

Divide the set of filters $F$ into $F^1, F^2,$ and $F^{1,2}$. For $i = 1, 2$, $F^i$ consists of those filters that may be applied either on $S_i$ before the join or on the join result after the join (denote $|F^i|$ by $n_i$). $F^{1,2}$ can be applied only on the join result. We assume that the join and the filters are independent, i.e., the selectivity of any operator does not depend on the operators applied earlier. Unlike the filters-only case, we do not have an extension, as we did
in Section 4.3.3, for the case when operator selectivities may be correlated. Our algorithm fails to extend because in the presence of a join, it is open whether the result in [98] holds, i.e., whether or not ordering the filters by conditional rank yields an approximation ratio of 4. When operator selectivities are independent, we have the following result (similar to Theorem 5.4 in [36]).

**Theorem 4.4.1.** Given join cost of the form \((4.10)\), in the optimal in-network query plan for \((4.9)\), the filters in \(\mathcal{F}^1\) (or \(\mathcal{F}^2\)) must be executed in rank order.

**Proof.** Let \(F_1\) and \(F_2\) be two filters in \(\mathcal{F}^1\), and \(\text{rank}(F_1) < \text{rank}(F_2)\). Suppose for contradiction that in the optimal plan \(P_{\text{opt}}\), \(F_2\) is executed (at node \(N_i\)) before \(F_1\) (at node \(N_{i+x}\), \(x \geq 0\)) as shown in Figure 4.9(a). Then in \(P_{\text{opt}}\), the join operator must have been executed between \(F_2\) and \(F_1\). Otherwise we could apply algorithm OPT_FILTER to the subplan starting at \(F_2\) and ending at \(F_1\), thus obtaining a lower-cost plan in which \(F_1\) and \(F_2\) are executed in rank order (according to line 9 of algorithm OPT_FILTER).

Let \(P_{\text{left}}, P_{\text{mid}}, P_{\text{right}}\) be the subplans before filter \(F_2\), between filters \(F_2\) and \(F_1\), and after filter \(F_1\), respectively. Let \(r_1\) and \(r_2\) be the rates of streams \(S_1\) and \(S_2\) on entering \(P_{\text{mid}},\)
and $r_{out}$ be the rate of the joined stream on exiting $P_{mid}$. Then $r_{out} = r_1 r_2 s$ where $s$ is a constant depending on the selectivity of the filters and the join operator in $P_{mid}$. Since filter costs and transmission costs are both linear in the stream rate, and the join cost is as given by (4.10), the total cost of $P_{mid}$ is $b_1 r_1 + b_2 r_2 + b_{12} r_1 r_2$ where $b_1, b_2, b_{12}$ are constants depending upon the costs of the filters and the join in $P_{mid}$. Let $\gamma$ be the scaledown in computational cost from node $N_i$ to node $N_{i+x}$. Then the total cost of $P_{opt}$ (excluding the costs of $P_{left}$ and $P_{right}$) is:

$$r_1 s(F_2) \leq c(F_2, i) + r_1 (b_1 + b_{12} r_2) + b_2 r_2 + r_1 r_2 s \gamma c(F_1, i)$$

(4.11)

Now consider two variations $P_1$ and $P_2$ of $P_{opt}$ as shown in Figures 4.9(b) and 4.9(c) where either $F_1$ or $F_2$ has been moved adjacent to the other, all other operators remaining the same. For both $P_1$ and $P_2$, the rate of stream $S_2$ entering $P_{mid}$ remains the same as in $P_{opt} (= r_2)$ since $F_1$ and $F_2$ operate only on stream $S_1$. However, the rate of stream $S_1$ entering $P_{mid}$ is $s(F_1) r_1$ for $P_1$ and $\frac{r_1 r_2 s \gamma c(F_1, i)}{s(F_2)}$ for $P_2$. Excluding the costs of $P_{left}$ and $P_{right}$ since they remain the same as in $P_{opt}$, the cost of $P_1$ is:

$$\frac{r_1}{s(F_2)} \cdot c(F_2, i) + r_1 c(F_1, i) + s(F_1) r_1 (b_1 + b_{12} r_2) + b_2 r_2$$

(4.12)

and that of $P_2$ is:

$$\frac{r_1}{s(F_2)} (b_1 + b_{12} r_2) + b_2 r_2 + r_1 r_2 s \gamma \left( \frac{c(F_2, i)}{s(F_2)} + c(F_1, i) \right)$$

(4.13)

Since $P_{opt}$ is optimal, we must have $(4.11)<(4.12)$ which on simplification gives:

$$r_2 s \gamma < 1 - \frac{1 - s(F_1)}{c(F_1, i)} (b_1 + b_{12} r_2)$$

(4.14)

Similarly, we must have $(4.11)<(4.13)$ which gives:

$$r_2 s \gamma > 1 - \frac{1 - s(F_2)}{c(F_2, i)} (b_1 + b_{12} r_2)$$

(4.15)

Combining (4.14) and (4.15) we get:

$$\frac{c(F_2, i)}{1 - s(F_2)} < \frac{c(F_1, i)}{1 - s(F_1)}$$
Algorithm \textit{OPT\_FILTER\_JOIN}

1. \( \mathcal{P}^* = \text{NULL}, c(\mathcal{P}^*) = \infty \)
2. for \( i = 0 \) to \( n_1 \), \( j = 0 \) to \( n_2 \), \( k = 1 \) to \( m \)
3. Construct new \( \mathcal{P} \) with join at node \( N_k \)
4. Optimally place \( F_{11}, \ldots, F_{1 i} \) at nodes \( N_1, \ldots, N_k \)
5. Optimally place \( F_{21}, \ldots, F_{2 j} \) at nodes \( N_1, \ldots, N_k \)
6. Optimally place remaining filters at \( N_k, \ldots, N_m \)
7. if \( (c(\mathcal{P}) < c(\mathcal{P}^*)) \) \( \mathcal{P}^* = \mathcal{P} \)
8. return \( \mathcal{P}^* \)

Figure 4.10: Operator Placement for Queries with Joins

which implies that \( \text{rank}(F_2) < \text{rank}(F_1) \), a contradiction.

Let \( F_{11}, \ldots, F_{n_11} \) be the filters in \( \mathcal{F}^1 \) in rank order. By Theorem 4.4.1, in the optimal plan there exists an \( i \) such that first \( F_{11}, \ldots, F_{1 i} \) are executed on stream \( S_1 \), followed by the join, and then \( F_{1i+1}, \ldots, F_{n_11} \) on the join result (similarly for \( \mathcal{F}^2 \)). Additionally, the join can be executed at each of the \( m \) nodes. Our algorithm (Figure 4.10) finds the optimal plan by an exhaustive search through these options. Lines 4-6 are each an invocation of algorithm \textit{OPT\_FILTER}. Hence the algorithm is polynomial, and a simple implementation runs in \( O(n_1n_2m(n + m)\log(n + m)) \) time.

4.5 Extensions

In this section, we define some interesting variations of the basic filter placement problem as future work, and we demonstrate their hardness.

4.5.1 Constrained Nodes

In some scenarios we may have constraints on the total amount of filter execution and transmission cost that certain nodes can incur. Given cost constraints at each node, a new problem is to find a feasible in-network query plan that satisfies these constraints, if such a plan exists. Recall the definition of \( c(\mathcal{P}, i) \) from Section 4.3.1.

\textbf{Definition 4.5.1 (Feasible Operator Placement).} Given cost constraint \( C_i \) at node \( N_i \), find an in-network query plan \( \mathcal{P} \) such that \( c(\mathcal{P}, i) \leq C_i \) for each \( i \) or return \( \text{NO} \) is no such \( \mathcal{P} \) exists.
Chapter 4. Operator Placement for In-Network Query Processing

Theorem 4.5.2. The feasible operator placement problem is NP-hard.

Proof. The proof is by a reduction from the PARTITION problem which is known to be NP-hard [63]. The PARTITION problem is to decide, given a set of numbers $S = \{a_1, \ldots, a_n\}$, whether there exists a subset $S'$ of $S$ such that $\sum_{a_i \in S'} a_i = \sum_{a_i \notin S'} a_i$. Based on such a given instance of PARTITION, we can construct an equivalent instance of the feasible operator placement problem. Construct $m = 2$ nodes and $n$ filters $F_1, \ldots, F_n$ with $c(F_i, 1) = a_i$. Let the computational cost be equal at both nodes, i.e., $\gamma_1 = 1$, and the transmission cost between the nodes be 0, i.e., $l_1 = 0$. Let the cost constraints at the nodes be $C_1 = C_2 = \sum_{i=1}^{n} a_i/2$. It is easily seen that a feasible operator placement for this instance exists iff the desired subset $S'$ for the PARTITION problem exists. This polynomial time reduction shows that the feasible operator placement problem is NP-hard.

In cost-constrained environments, a further desirable property might be load balancing: We might prefer a plan having overall higher cost if it places roughly equal load on each node, as compared to a plan that has lower cost but loads a few nodes very heavily or up to capacity. Load balancing may be particularly applicable when the system is required to support a number of concurrent queries.

Definition 4.5.3 (Load-Balancing). Given cost constraint $C_i$ at node $N_i$, find the in-network query plan $P$ that minimizes $\max_{1 \leq i \leq m} \{c(P, i)/C_i\}$.

Clearly, the load balancing problem is at least as hard as the feasible operator placement problem, since if $\max\{c(P, i)/C_i\} < 1$ we have found a feasible operator placement. Thus, the load balancing problem is NP-hard.

4.5.2 Per-Filter Cost Scaling

So far we have assumed that the cost of each filter scales down by a factor $\gamma_i$ from node $N_i$ to $N_{i+1}$. However, the cost of different filters may change differently from one node to the next, i.e., $c(F, i + 1)/c(F, i)$ may be different for different $F$. For example, if a filter $F$ accesses external data that resides close to node $N_i$, it may be more expensive to execute $F$ at node $N_{i+1}$ than at node $N_i$. Meanwhile, other filters may be cheaper at node $N_{i+1}$ simply because $N_{i+1}$ has higher computational power. When we have per-filter cost scaling, the technique we used of modeling network links as filters no longer applies. Whether the problem becomes NP-hard with per-filter cost scaling remains an open question.
4.6 Related Work

A considerable amount of work has focused on extending centralized data stream query processing systems [18] to a distributed setting, e.g., Borealis [5], HourGlass [104], Iris-Net [50], and NiagaraCQ [38]. Most of this work considers internet-style network topologies consisting of nodes with ample computational power. Consequently, the work focuses on optimizing network usage and minimizing latency, and is not concerned with computational overload. Even when computational overload is considered, e.g. in [39], only heuristics are provided to move load from one node to another.

Our work addresses the considerably different scenario of real-world monitoring applications [61], where optimization of both communication and computation is required. There has been some previous work on in-network processing in these environments, but it focuses primarily on aggregation [88], and has not considered expensive filters or joins. Acquisitional query processing [89] focusses on where, when, and how often data is physically acquired and delivered to the query operators, but the problem of operator placement is not dealt with.

In classical relational query optimization, filters are usually assumed to be inexpensive, and a common optimization heuristic is to push filters as close to the leaf operators as possible. Query optimization and site selection for distributed databases is also well studied [54,73], again assuming inexpensive filters. Expensive filters have been considered in the context of query optimization with user-defined predicates [36,76], but only in a centralized setting.

4.7 Conclusions

In this chapter, we addressed the problem of query processing in real-world monitoring applications that often exhibit progressively increasing computational power and network bandwidth up a hierarchy of processing nodes. Data is acquired at low-capability edge devices and transmitted up the hierarchy to the root, where queries are posed and results collected. To reduce communications costs, query operators can be executed lower in the hierarchy, typically at the expense of higher computational cost. We addressed the problem of balancing computational cost against network communication costs to obtain an optimal operator placement algorithm that minimizes overall cost. We showed that the
problem is tractable, but that a greedy algorithm can be suboptimal. We provided an optimal algorithm for uncorrelated filters, then extended our approach to correlated filters and multiway stream joins. Finally, we posed several interesting variations of our problem as open topics for future research.
Chapter 5

Query Optimization over Web Services

As described in Chapter 1, this thesis addresses problems pertaining to efficient query processing in two new data management scenarios. So far in the thesis, we have considered the first of these two scenarios: one in which data is continuously arriving in the form of streams. We now turn to the other modern data management scenario considered in this thesis: one in which data is residing behind restricted, but highly standardized, interfaces called web services (recall Section 1.2.2). In this chapter, we propose the architecture and component of a general-purpose Web Service Management System (WSMS) that enables querying of the data behind multiple web services in a transparent and integrated fashion. We then tackle a first basic WSMS problem: query planning for Select-Project-Join queries spanning multiple web services.

5.1 Introduction

Web services [126] have become the standard by which any data or functionality can be exported to the world over a network such as the web. Service-oriented architecture is common within enterprises with their databases being put behind web services, which provide an interoperable method of interacting with the data. This web-services paradigm is especially attractive since even data stored in formats other than traditional databases can be made available via web services. Consequently, there is a need for providing DBMS-like capabilities when data sources are web services, instead of traditional stored (or even
streamed) data. To this end we propose the development of a Web Service Management System (WSMS): a general-purpose system that enables clients to query multiple web services simultaneously in a declarative, transparent, and integrated fashion.

Overall, a WSMS consists of three major components; see Figure 5.1. The Metadata component deals with metadata management, registration of new web services, and mapping their schemas to an integrated view provided to the client. There is a large body of work on data integration, see e.g., [30, 96], that applies to the Metadata component; we do not focus on these problems. Given an integrated view of the schema, a client can query the WSMS through an SQL-like interface. The Query Processing and Optimization component handles optimization and execution of such declarative queries, i.e., it chooses and executes a query plan whose operators invoke the relevant web services. The Profiling and Statistics component monitors web services to determine their response time characteristics, and maintains relevant statistics over the web service data, to the extent possible. This component is used primarily by the query optimizer for making its optimization decisions. In this chapter, we primarily focus on the Query Processing and Optimization component, and address the problem of optimizing Select-Project-Join queries spanning multiple web services. In the next chapter, we give techniques for designing the Profiling and Statistics component.

Most web services provide a function-call like interface $X \rightarrow Y$ where $X$ and $Y$ are sets of attributes: given values for the attributes in $X$, the web service returns values for the attributes in $Y$. For example, a web service may take a credit card number and return the
5.1. INTRODUCTION

card’s credit limit. Due to this very restricted interface, most query processing over web services can be thought of in terms of a “workflow” or pipeline: some input data is fed to the WSMS, and the WSMS processes this data through a sequence of web services. The output of one web service is returned to the WSMS and then serves as input to the next web service in the pipeline, finally producing the query results. Each web service in the pipeline typically performs operations such as filtering out data items that are not relevant to the query, transforming data items, or appending additional information to each data item. Transformed or augmented data items may be required for further processing of the query (effectively performing a join across web services), or may become a part of the final query result.

Example 5.1.1. Suppose a credit card company wishes to send out mailings for its new credit card offer. The company continuously obtains lists of potential recipients from which it wants to select only those who have a good payment history on a prior credit card, and who have a credit rating above some threshold. For processing this query, the company has the following three web services at its disposal.

\[ WS_1 : \text{name (n)} \rightarrow \text{credit rating (cr)} \]
\[ WS_2 : \text{name (n)} \rightarrow \text{credit card numbers (ccn)} \]
\[ WS_3 : \text{card number (ccn)} \rightarrow \text{payment history (ph)} \]

With a WSMS, one possible way of executing the query is as follows: The company’s initial list of names (we assume names are unique) is first processed by \( WS_1 \) to determine the corresponding credit ratings, and those below threshold are filtered out (either by \( WS_1 \) itself or by the WSMS). The remaining names are then processed by \( WS_2 \) to get the corresponding credit card numbers. Each card number is then processed by \( WS_3 \), and if the card is found to have a good payment history, then the name is output in the result of the query.

\[ \square \]

Challenges

The first obvious step to speed up query execution in a WSMS is to use the conventional idea of pipelined parallelism: data already processed by web service \( WS_i \) may be processed by a subsequent web service \( WS_{i+1} \) in the pipeline, at the same time as \( WS_i \) processes new data. Deciding the optimal way to perform this pipelining poses several new challenges:

1. Different web services may differ widely in their response time characteristics, as
well as in how many output tuples they produce per input tuple on average (henceforth \textit{selectivity}). Hence different arrangements of the web services in the pipeline may result in significantly different overall processing rates. The optimizer must decide the best arrangement, or \textit{execution plan}, for a given query.

2. The web services in the pipeline may not always be freely reordered, i.e., there might exist \textit{precedence constraints}. (In Example 5.1.1, WS$_2$ must occur before WS$_3$ in the pipeline.) In such cases, the optimizer must pick the best plan that respects all precedence constraints.

3. A linear ordering of the web services in a pipeline (as in Example 5.1.1) may not be optimal. For example, if there are no precedence constraints between web services WS$_i$ and WS$_j$, we need not wait for results from one to invoke the other, rather they may be invoked in parallel using the same input data. On the other hand, parallelizing all web services without precedence constraints may not be optimal either, since one or more of the web services may vastly reduce the amount of data the others need to process.

4. Each web service call usually has some fixed overhead, typically parsing SOAP/XML headers and going through the network stack. Hence some web services support sending data to them in “chunks” rather than one tuple at a time. Through experiments we found that the response time of a web service often is not linear in the input chunk size, so the optimizer must decide the best chunk size to use.

In this chapter, we develop new, efficient algorithms that address each of the above challenges to arrive at the optimal pipelined execution plan for a given query over a set of web services. A simple yet significant observation that forms the basis for our algorithms is that the performance of a pipelined plan over web services (the rate of data processing through the plan) is dictated by the slowest web service in the pipeline (referred to as the \textit{bottleneck cost metric}). In contrast, in a traditional centralized system, the cost of a pipelined plan is dictated by the sum of the costs of the plan operators (referred to as the \textit{sum cost metric}) rather than by the cost of only the slowest operator. Previous related work [22,36,76,78] has considered only the sum cost metric. To the best of our knowledge, our work is the first to consider the bottleneck metric.
5.1. INTRODUCTION

We start by considering web services without precedence constraints and give a simple algorithm to find the optimal plan based on the web service response times and selectivities. Our algorithm reveals the somewhat counterintuitive property that when the selectivity of all web services is \( \leq 1 \), the optimal arrangement depends only on the response times of the web services and is independent of their selectivities.

Next we give a polynomial-time algorithm to find the optimal plan when there may be arbitrary precedence constraints among the web services. It is surprising that such an algorithm exists, since under the sum cost metric, it is known that the optimal plan is polytime computable only for restricted types of precedence constraints \([78]\), and for arbitrary precedence constraints the optimal plan is hard to even approximate \([29]\).

We then consider sending data to web services in chunks. We show that our query optimization algorithm extends trivially to account for chunking. We also give an algorithm to determine the best chunk size to use for each web service. The algorithm is based on profiling the web services to determine their response times as a function of the size of the data chunk sent to them. Finally, we report on experiments with an initial prototype that indicate that our algorithms can lead to significant performance improvement over more straightforward techniques.

Chapter Organization

The rest of the chapter is organized as follows. In Section 5.2, we formally define the class of queries we consider, introduce the model for query processing in a WSMS, and formalize the bottleneck cost metric that is used to compare query plans. Next, we give algorithms to decide the best arrangement of web services into a pipelined plan so that the overall processing rate is maximized, both in the absence of precedence constraints (Section 5.3), and in the presence of arbitrary precedence constraints (Section 5.4). In Section 5.5, we consider the case when data can be sent to web services in chunks, and we give an algorithm to decide the optimal data chunk size for each web service in a query plan. In Section 5.6, we report an experimental evaluation of our algorithms with an initial prototype WSMS query optimizer (with simple instantiations of the other WSMS components in Figure 5.1). Finally, we discuss related work in Section 5.7, and conclude in Section 5.8.
5.2 Preliminaries

Consider a WSMS as shown in Figure 5.1 that provides an integrated query interface to \( n \) web services \( \text{WS}_1, \ldots, \text{WS}_n \). We assume that for querying, each web service \( \text{WS}_i \) provides a function-call like interface \( X_i \rightarrow Y_i \), i.e., given values for attributes in \( X_i \), the web service returns values for the attributes in \( Y_i \). Using the notation of binding patterns [60], we write \( \text{WS}_i(X_{i}^b, Y_{i}^f) \) to denote that, treating \( \text{WS}_i \) as a virtual table, the values of attributes in \( X_i \) must be specified (or bound) while the values of attributes in \( Y_i \) are retrieved (or free).

Let \( \bar{x} \) and \( \bar{y} \) denote value assignments to the attributes in \( X_i \) and \( Y_i \) respectively. Logically, virtual table \( \text{WS}_i \) has a tuple \( (\bar{x}, \bar{y}) \) whenever the value assignment \( \bar{y} \) is among those returned for \( Y_i \) when \( \text{WS}_i \) is invoked with \( X_i = \bar{x} \). There may be zero, one, or many tuples in \( \text{WS}_i \) for each possible \( \bar{x} \). Note that if \( Y_i \) is empty, then web service \( \text{WS}_i \) acts as a filter, and virtual table \( \text{WS}_i \) contains a tuple \( \bar{x} \) for every value assignment \( \bar{x} \) passing the filter.

5.2.1 Class of Queries Considered

The class of queries we consider for optimization are Select-Project-Join (SPJ) queries over one or more web services \( \text{WS}_1, \ldots, \text{WS}_n \) and a table \( I \) corresponding to data input by the client to the WSMS (e.g., the initial set of names in Example 5.1.1). We assume that the correspondence among various attributes of various web services, required for joins, is tracked by the Metadata component of the WSMS (Figure 5.1).

**Definition 5.2.1 (SPJ Queries over Web Services).**

\[
\begin{align*}
\text{SELECT} & \quad A_s \\
\text{FROM} & \quad I(A_I) \Join \text{WS}_1(X_{1}^b, Y_{1}^f) \Join \ldots \Join \text{WS}_n(X_{n}^b, Y_{n}^f) \\
\text{WHERE} & \quad P_1(A_1) \land P_2(A_2) \land \ldots \land P_m(A_m)
\end{align*}
\]

where \( A_s \) is the set of projected attributes, \( A_I \) is the set of attributes in the input data, and \( P_1, \ldots, P_m \) are predicates applied on attributes \( A_1, \ldots, A_m \) respectively.

We assume in Definition 5.2.1 that all predicates are on single attributes, i.e., there are no join conditions except implicit natural equijoins. We also assume there is only one source of information for each attribute: each attribute is either specified in the input data, or is obtained as a free attribute from exactly one web service. (When values are available from multiple web services it becomes important to address issues such as deciding which web service is of higher quality, which we have not yet addressed.)
5.2 PRELIMINARIES

Definition 5.2.2 (Precedence Constraints). If a bound attribute in $X_j$ for WS$_j$ is obtained from some free attribute $Y_i$ of WS$_i$, then there exists a precedence constraint WS$_i$ $\prec$ WS$_j$, i.e., in any feasible execution plan for the query, WS$_i$ must precede WS$_j$.

The precedence constraints may be represented as a directed acyclic graph (DAG) $G$ in which there is a node corresponding to each web service, and there is a directed edge from WS$_i$ to WS$_j$ if there is a precedence constraint WS$_i$ $\prec$ WS$_j$. Note that $G$ is not specified by the query, it is implied by which attributes are bound and which ones are free in the web services involved in the query.

Example 5.2.3. We continue with Example 5.1.1. With binding patterns, the three web services can be expressed as WS$_1$($n^b$, cr$^f$), WS$_2$($n^b$, ccn$^f$), and WS$_3$($ccn^b$, ph$^f$). Denoting the input names by $I$, the example query can be expressed as:

```
SELECT n
FROM I($n$) $\bowtie$ WS$_1$($n^b$, cr$^f$) $\bowtie$ WS$_2$($n^b$, ccn$^f$) $\bowtie$ WS$_3$($ccn^b$, ph$^f$)
WHERE cr $>$ threshold $\land$ ph = good
```

Since the bound attribute ccn in WS$_3$ is provided by WS$_2$, there exists a precedence constraint WS$_2$ $\prec$ WS$_3$.

5.2.2 Query Plans and Execution Model

We motivate possible execution plans through the following example.

Example 5.2.4. We continue with Example 5.1.1. The execution plan discussed in Example 5.1.1 is shown by Plan 1 in Figure 5.2. Although we show direct arrows between web services, in reality the arrows imply that data is returned to the WSMS, relevant predicates are applied, and the results are passed to the next web service.

However, since there is no precedence constraint between WS$_1$ and WS$_2$, we need not wait for output from WS$_1$ to invoke WS$_2$. Thus, an alternative execution plan for the same query is shown...
by Plan 2 in Figure 5.2, where the input list of names \( I \) is dispatched in parallel to WS\(_1\) and WS\(_2\) (denoted by two outgoing arrows from \( I \)). The results from WS\(_2\) are then used to invoke WS\(_3\) as in Plan 1. The final query result is obtained by joining the results from the two branches locally at the WSMS.

In general, an execution plan is an arrangement of the web services in the query into a DAG \( H \) with parallel dispatch of data denoted by multiple outgoing edges from a single web service, and rejoining of data denoted by multiple incoming edges into a web service. Note that the plan DAG \( H \) is distinct from the DAG \( G \) of precedence constraints among web services, although if WS\(_i\) is an ancestor of WS\(_j\) in \( G \), then it must also be so in \( H \) (i.e., \( H \) respects the precedence constraints specified by \( G \)).

Given a plan DAG \( H \), it is executed by the WSMS as follows (see Figure 5.3). A thread \( T_i \) is established for each web service WS\(_i\). Thread \( T_i \) takes input tuples from a separate join thread \( J_i \) that joins the outputs of the parents of WS\(_i\) in \( H \). In the special cases when WS\(_i\) has no parents in \( H \), \( T_i \) takes input from the table \( I \), and when WS\(_i\) has exactly one parent in \( H \) (say WS\(_p\)), \( T_i \) takes input directly from the output of thread \( T_p \). Thread \( T_i \) uses its input tuples to invoke WS\(_i\), filters the returned tuples, and writes them to its output. The final query result is obtained from the output of a join thread \( J_{out} \) that performs a join of the outputs of all the web services that are leaves in the plan \( H \). In the special case when there is only one leaf web service WS\(_l\) in \( H \) (e.g., Plan 1 in Figure 5.2), the output from WS\(_l\) directly forms the query result and thread \( J_{out} \) is not needed.

The join threads perform a multiway stream join of their inputs, and there are known techniques to perform such joins efficiently, e.g., [125]. Furthermore, using a technique similar to punctuations in data streams [52], a unique marker is inserted when branching, so the join threads know when joining tuples are complete and state can be discarded.

According to Figure 5.3, the WSMS can have only one outstanding call to any individual web service at a time, i.e., while the WSMS is waiting for the results from a previous call to WS\(_i\) to arrive, it cannot make another call to WS\(_i\). However, this assumption is not important to our approach. As we will see, our algorithms only rely on a quantity representing the maximum rate at which a web service can process data. This rate can often be boosted by making multiple outstanding calls to the web service [68]; how the rate is achieved does not affect the applicability of our algorithms.

Also, we assume that results from one web service are returned to the WSMS before being passed on to another web service. However, with sufficient standardization, it might be
5.2. PRELIMINARIES

Algorithm \textit{ExecutePlan}(\mathcal{H})

\mathcal{H}: An arrangement of web services into a DAG

1. for each web service WS
2. launch a thread \( T_i \)
3. if WS has no parents in \( \mathcal{H} \)
   4. set up \( T_i \) to take input from \( I \)
   5. else if WS has a single parent WS\_p in \( \mathcal{H} \)
   6. set up \( T_i \) to take input from \( T_p \)'s output
   7. else
   8. launch a join thread \( J_i \)
   9. set up \( T_i \) to take input from \( J_i \)'s output
   10. launch join thread \( J_{out} \)
11. return output of \( J_{out} \) as query result

\textbf{Thread} \( T_i \):

1. while (tuples available on \( T_i \)'s input)
2. read a tuple \( s \) from \( T_i \)'s input
3. invoke WS with values \( s . X_i \)
4. for each returned tuple \( t \)
5. apply all predicates \( P_j (A_j) \) for \( A_j \in \mathcal{Y}_i \)
6. if \( t \) satisfies all predicates
7. write \( s \uplus t \) to \( T_i \)'s output

\textbf{Thread} \( J_i \):

1. perform join of WS\_i's parents' output

\textbf{Thread} \( J_{out} \):

1. perform join of the outputs of web services that are leaves in \( \mathcal{H} \)

Figure 5.3: Query Execution Algorithm

possible for one web service to send it results directly to another. Our model for pipelined execution and our optimization algorithms would not change under that model.

5.2.3 Bottleneck Cost Metric

As a first step in developing a query optimizer for web services, we assume that the goal of optimization is to minimize the total running time of queries. In reality, there are other metrics that might also be important. For example, if a web service call incurs a monetary cost, we may wish to minimize the total number of web service calls. A study of other interesting metrics and their tradeoff with query running time is an interesting topic of future work.

To obtain an expression for the running time of a particular query execution plan, we assume that the following two quantities can be tracked and estimated for each web service by the Profiling and Statistics component of the WSMS (Figure 5.1); we will give some profiling techniques in the next chapter.

1. **Per-tuple Response time** \( (c_i) \): If \( r_i \) is the maximum rate at which results of invocations can be obtained\(^1\) from WS\(_i\), we use \( c_i = 1/r_i \) as the effective per-tuple response time (or intuitively, the cost) of WS\(_i\). The maximum rate \( r_i \) for a web service WS\(_i\)

\(^1\)Note that \( r_i \) incorporates the time for transmission over the network, as well as the queuing delays and the processing time at the web service.
(and hence its effective per-tuple response time $c_i$) can often be boosted by batching several calls together (data chunking, Section 5.5), or making multiple parallel calls (as described at the end of Section 5.2.2). Our optimization algorithms are applicable regardless of how the best response time for a web service is achieved.

The response time of a web service may depend on a variety of factors, such as the web service provisioning, the load on the web service, and the network conditions. As a first step, we give algorithms assuming the response time is a constant, so the query may need to be reoptimized if significant changes in response time are detected. As future work, we plan to explore ideas such as adaptive plans, or plans that are provably robust to variations in response time.

2. **Selectivity** ($s_i$): Recall lines 2-7 of thread $T_i$ (Figure 5.3) where a tuple is input to $WS_i$ and relevant filters are applied to the returned results. The average number of returned tuples (per tuple input to $WS_i$) that remain unfiltered after applying all relevant predicates is denoted by $s_i$, and is referred to as the *selectivity* of web service $WS_i$. $s_i$ may be $\leq 1$. For instance, in Example 5.2.3, if 10% of the names in $I$ have credit rating above threshold, then $s_1 = 0.1$. In general, $s_i$ may also be $> 1$. In the same example, if every person holds 5 credit cards on average, then $s_2 = 5$.

We assume web service selectivities are *independent*, i.e., the selectivity of a web service does not depend on which web services have already been invoked. Extending our algorithms to work with *correlated* selectivities is an important direction for future work. Although, our model of selectivities is fairly general, it is not adequate to capture scenarios where the web service performs some form of aggregation, i.e., producing a fixed number of output tuples irrespective of the number of input tuples. Extension of our algorithms to such web services is also an item of future work.

Consider the pipelined execution of a plan as specified in Figure 5.3. There is a time period at the beginning (respectively end) of query execution when the pipeline is filling up (respectively emptying out), after which a steady state is reached during which input tuples flow through the pipeline at a constant rate. For long-running queries—typically queries in which the input table $I$ is large—the time spent to reach steady state is negligible compared to the total running time of the query. In such cases, minimizing the total running time is equivalent to maximizing the rate at which tuples in $I$ are processed through the pipeline in steady state. When time to reach steady state is nonnegligible,
then the query is typically short-running and less in need of optimization anyway. Thus, we focus on the processing rate during steady state.

Since all web services can be executing in parallel, the maximum rate at which input tuples can be processed through the pipelined plan is determined by the bottleneck web service: the web service that spends the most time on average per original input tuple in I. Next, we derive a formal expression for this cost metric.

Consider a query plan $\mathcal{H}$ specified as a DAG on the web services in the query. Let $P_i(\mathcal{H})$ denote the set of predecessors of WS$_i$ in $\mathcal{H}$, i.e., all web services that are invoked before WS$_i$ in the plan. Formally,

$$P_i(\mathcal{H}) = \{\text{WS}_j \mid \text{WS}_j \text{ has a directed path to } \text{WS}_i \text{ in } \mathcal{H}\} \quad (5.1)$$

Given a set $\mathcal{S}$ of web services, we define the combined selectivity of all the web services in $\mathcal{S}$ as $R[\mathcal{S}]$. By independence, $R[\mathcal{S}]$ is given by:

$$R[\mathcal{S}] = \prod_{i \mid \text{WS}_i \in \mathcal{S}} \sigma_i \quad (5.2)$$

Then, for every tuple in $I$ input to plan $\mathcal{H}$, the average number of tuples that WS$_i$ needs to process is given by $R[P_i(\mathcal{H})]$. Since the average time required by WS$_i$ to process a tuple in its input is $c_i$, the average processing time required by web service WS$_i$ (or intuitively, the cost incurred by WS$_i$) per original input tuple is $R[P_i(\mathcal{H})] \cdot c_i$. Recall that plan cost is determined by the web service with maximum processing time per original input tuple. Thus the cost of the query plan $\mathcal{H}$ is given by the following metric (referred to as the bottleneck cost metric):

$$\text{cost}(\mathcal{H}) = \max_{1 \leq i \leq n} \left( R[P_i(\mathcal{H})] \cdot c_i \right) \quad (5.3)$$

**Example 5.2.5.** Consider Plan 1 in Figure 5.2 for the query in Example 5.2.3. Let the costs and selectivities of the web services be as follows:

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost of WS$_i$ ($c_i$)</td>
<td>2</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Selectivity of WS$_i$ ($s_i$)</td>
<td>0.1</td>
<td>5</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Let $|I|$ be the number of tuples in the input data $I$. In Plan 1, with the example selectivities, WS$_1$ needs to process $|I|$ tuples, WS$_2$ needs to process $0.1|I|$ tuples, and WS$_3$ needs to process $0.5|I|$.
tuples. Thus, the time taken by WS\textsubscript{1}, WS\textsubscript{2} and WS\textsubscript{3} per tuple in I is 2, 1, and 2.5 respectively. The cost of the plan is then \(\text{max}(2, 1, 2.5) = 2.5\). We arrive at the same number using formula (5.3).

Now consider Plan 2 in Figure 5.2. Its cost (using (5.3)) is \(\text{max}(10, 25, 2) = 25\). Thus, for this example, Plan 2 is 10 times slower than Plan 1.

It may appear that the bottleneck cost metric ignores the work that must be done by the WSMS threads (Figure 5.3). Formally, we can treat all the work done at the WSMS as just another call in the pipeline. Our algorithms are designed under the assumption that the pipeline stage constituted by the WSMS is never the bottleneck, which seems realistic since it is unlikely that the simple operations the WSMS needs to perform will be more expensive than remote web service calls. This assumption is also validated by our experiments (Section 5.6).

We can now formally define the query optimization problem solved in this chapter.

**Definition 5.2.6.** (Query Optimization Over Web Services). Given an SPJ query over web services (Definition 5.2.1) implying a DAG \(\mathcal{G}\) of precedence constraints, find a query plan arranging the web services into a DAG \(\mathcal{H}\) that respects all precedence constraints in \(\mathcal{G}\), where cost(\(\mathcal{H}\)) as given by (5.3) is minimized.

It is important to understand the basic differences between our scenario and a traditional centralized setting which also has query operators characterized by costs and selectivities. In the traditional setting, each operator is running on the same machine, so the cost of the plan is given not by the bottleneck cost but by the sum of the costs incurred by the operators (referred to as the sum cost metric). The sum cost metric has been considered in much previous work [22, 36, 76, 78], but to the best of our knowledge, our work is the first to consider the fundamentally different bottleneck cost metric. One critical difference between the two metrics as brought out in this chapter is that under the bottleneck cost metric, the optimal plan can be found in polynomial time for general precedence constraints (as shown in Section 5.4), while under the sum cost metric, for general precedence constraints the optimal plan is hard to even approximate in polynomial time [29, 78].
5.3 No Precedence Constraints

In this section, we consider the special case of the problem given by Definition 5.2.6 when there are no precedence constraints, i.e., the DAG $\mathcal{G}$ has no edges. The absence of precedence constraints implies that no web service depends on another for its bound attributes, i.e., all bound attributes are available directly from the input data $I$. Then, a simple execution plan is to dispatch the input $I$ in parallel to each of the web services, with the results joined back at the WSMS.

The main problem with simultaneously dispatching $I$ to all of the web services is simply that each web service must process all of the tuples in $I$. If some web services are selective (i.e., have selectivity $\leq 1$), then it is better for the slower web services to come near the end of the pipeline, reducing how much data they must process. This basic observation forms the intuition behind our algorithm for selective web services (Section 5.3.1).

When web services may be proliferative (i.e., have selectivity $> 1$), we do use the idea of dispatching input in parallel to multiple web services. One interesting observation in our results is that the optimal arrangement of web services depends only on whether their selectivity is $\leq 1$ or $> 1$, but not on the exact selectivity value.

5.3.1 Selective Web Services

In this section, we focus on the case when there are no precedence constraints and the selectivity of each web service is $\leq 1$. Our algorithm for selective web services follows directly from the following two simple but key observations.

**Lemma 5.3.1.** There exists an optimal plan that is a linear ordering of the web services, i.e., has no parallel dispatch of data.

**Proof.** Suppose in the optimal plan $\mathcal{H}$ there is parallel dispatch of data to $WS_i$ and $WS_j$. Modify $\mathcal{H}$ to $\mathcal{H}'$ where $WS_i$ is moved to the point before the parallel dispatch, and the rest of $\mathcal{H}$ remains unchanged. The amount of data seen by each web service in $\mathcal{H}'$ is either the same as in $\mathcal{H}$, or $s_i$ times that in $\mathcal{H}$. Since $s_i \leq 1$, the bottleneck in $\mathcal{H}'$ is at most as much in $\mathcal{H}$. Continuing, this flattening we can find a linear plan that has cost at most that of $\mathcal{H}$. \(\square\)

**Lemma 5.3.2.** Let $WS_1, \ldots, WS_n$ be a plan with a linear ordering of the web services. If $c_i > c_{i+1}$, then $WS_i$ and $WS_{i+1}$ can be swapped without increasing the cost of the plan.
Proof. Let \( \mathcal{H} \) be the plan in which the ordering of the web services is \( \text{WS}_1, \ldots, \text{WS}_n \), and let \( \mathcal{H}' \) denote the same plan but with \( \text{WS}_i \) swapped with \( \text{WS}_{i+1} \). Let \( f \) denote \( \prod_{j=1}^{i-1} s_j \). By (5.3):

\[
\text{cost}(\mathcal{H}) = \max(f c_i, f s_i c_{i+1}, \text{other terms})
\]

where the other terms are the cost terms for the rest of the web services. These other terms remain the same when we consider \( \mathcal{H}' \). Thus:

\[
\text{cost}(\mathcal{H}') = \max(f c_{i+1}, f s_{i+1} c_i, \text{other terms})
\]

Consider the terms in \( \text{cost}(\mathcal{H}') \). \( f c_{i+1} < f c_i \) by the lemma statement, and \( f s_{i+1} c_i \leq f c_i \) since \( s_{i+1} \leq 1 \). Since other terms in \( \text{cost}(\mathcal{H}') \) are also present in \( \text{cost}(\mathcal{H}) \), \( \text{cost}(\mathcal{H}') \leq \text{cost}(\mathcal{H}) \).

Lemmas 5.3.1 and 5.3.2 immediately lead to the following result.

**Theorem 5.3.3.** For selective web services with no precedence constraints, the optimal plan is a linear ordering of the web services by increasing response time, ignoring selectivities.

**Proof.** From Lemma 5.3.1, there exists a linear ordering of the web services that is optimal. Consider any linear ordering of the web services that is optimal. If, in this ordering, there is a higher cost service followed immediately by a lower cost service, by Lemma 5.3.2 we can swap them without increasing the cost of the plan. We continue such swapping until there does not exist a higher cost web service followed immediately by a lower cost one, thereby obtaining the result.

Recall that in the derivation of the cost expression for plans (Section 5.2.3), we assumed that the selectivities of web services are independent. If independence does not hold, the cost of the query plan can be written in terms of conditional rather than absolute selectivities. However, as long the conditional selectivities are also \( \leq 1 \), Theorem 5.3.3 applies. Thus our result extends to web services with correlated selectivities.

### 5.3.2 Proliferative Web Services

We now consider the case when some web services may have selectivity \( > 1 \).

**Theorem 5.3.4.** Figure 5.4 depicts the overall optimal plan for a query consisting of both selective and proliferative web services with no precedence constraints.
5.4. PRECEDENCE CONSTRAINTS

Proof. Consider a query consisting of a set of selective web services \( \mathcal{W}_s \) and a set of proliferative web services \( \mathcal{W}_p \), and having no precedence constraints. In the absence of precedence constraints, a web service \( WS_i \in \mathcal{W}_p \) should not occur before any other web service \( WS_j \) in a pipeline, since it will only increase work for \( WS_j \). Thus, the web services in \( \mathcal{W}_p \) should be invoked in parallel at the end of the plan. Using the results from Section 5.3.1, the web services in \( \mathcal{W}_s \) should be placed in increasing cost order. Thus, the overall optimal plan is as shown in Figure 5.4.

5.4 Precedence Constraints

In this section, we develop a general, polynomial-time algorithm for the problem given by Definition 5.2.6 when there may be precedence constraints among some of the web services in the query (recall Definition 5.2.2). Recall that precedence constraints are specified as a DAG \( G \).

We define \( M_i \) as the set of all web services that are prerequisites for \( WS_i \), i.e.,

\[
M_i = \{ WS_j \mid WS_j \prec WS_i \}
\]

Our overall algorithm (described in Section 5.4.3) builds the plan DAG \( \mathcal{H} \) incrementally by greedily augmenting it one web service at a time. At any stage the web service that is chosen for addition to \( \mathcal{H} \) is the one that can be added to \( \mathcal{H} \) with minimum cost, and all of whose prerequisite web services have already been added to \( \mathcal{H} \). The crux of the algorithm lies in finding the minimum cost of adding a web service to \( \mathcal{H} \), described next.
5.4.1 Adding a Web Service to the Plan

Suppose we have constructed a partial plan $\tilde{H}$ and we wish to add $WS_x$ to $\tilde{H}$. To find the minimum cost of adding $WS_x$ to $\tilde{H}$ (without modifying $\tilde{H}$), we compute the best cut $C_x$ in the DAG $\tilde{H}$, such that on placing edges from the web services in $C_x$ to $WS_x$, the cost incurred by $WS_x$ is minimized. An example of a cut is shown in Figure 5.5. Formally, a cut $C$ in a DAG $\mathcal{H}$ is defined as any set of web services in $\mathcal{H}$ such that there does not exist a directed path in $\mathcal{H}$ from one web service in the cut to another. For the cut $C_x$ in $\tilde{H}$, we also define the set $P_{C_x}$ (also shown in Figure 5.5) as consisting of all the web services in $C_x$ and all their predecessors in $\tilde{H}$, i.e., (recall (5.1) for definition of $P_j(\tilde{H})$)

$$P_{C_x} = C_x \cup \{WS_i \mid WS_i \in P_j(\tilde{H}) \text{ for } WS_j \in C_x\}$$  \hspace{1cm} (5.7)

Note that given $P_{C_x}$, the cut $C_x$ can be easily computed as only those web services in $P_{C_x}$ that are not predecessors of some other web service in $P_{C_x}$.

Recall definition of $R[S]$ from (5.2). When we place edges from the web services in $C_x$ to $WS_x$ (as shown in Figure 5.5), the total cost incurred by $WS_x$ is given by:

$$\text{cost}(WS_x) = R[P_{C_x}] \cdot c_x$$

Let us associate a variable $z_i$ with every $WS_i \in \tilde{H}$ that is set to 1 if $WS_i \in P_{C_x}$, and to 0 otherwise. Then, from (5.2), we have:

$$R[P_{C_x}] = \prod_{i \mid WS_i \in \tilde{H}} (\sigma_i)^{z_i}$$  \hspace{1cm} (5.8)

Then the optimal set $P_{C_x}$ (and hence the optimal cut $C_x$) such that cost($WS_x$) is minimized, is obtained by solving the following linear program where the variables are the $z_i$s.

| Minimize $\log c_x + \sum_{i \mid WS_i \in \tilde{H}} z_i \log \sigma_i$ subject to |
| $z_i = 1$ \hspace{1cm} $\forall i \mid WS_i \in M_x$ |
| $z_i \geq z_j$ \hspace{1cm} $\forall i, j \mid WS_i \in P_j(\tilde{H})$ |
| $z_i \in [0, 1]$ \hspace{1cm} $\forall i$ |

The objective function of the above linear program minimizes $\log(\text{cost}(WS_x))$ that is
5.4. PRECEDENCE CONSTRAINTS

equivalent to minimizing cost(WS_x). We take logarithms to ensure that the objective function is linear in the variables. The first constraint in (5.9) ensures that \( P_{C_x} \) includes all the prerequisite web services for WS_x (so that it is feasible to add WS_x after the cut \( C_x \)). The second constraint ensures that the set \( P_{C_x} \) is chosen according to the current structure of \( \tilde{H} \), i.e., if a web service WS_i is chosen in \( P_{C_x} \), all predecessors of WS_i in \( \tilde{H} \) are also chosen in \( P_{C_x} \). Note that the third constraint relaxes the linear program to include fractional \( z_i \)'s instead of just integers. However, there always exists an optimal integer solution to the above linear program as shown by the following theorem.

**Lemma 5.4.1.** The linear program (5.9) has an optimal solution where each \( z_i \) is set to either 0 or 1.

**Proof.** Consider any optimal solution to the linear program (5.9). Choose \( r \) uniformly at random in the range \([0, 1]\). If \( z_i > r \), set \( z_i \) to 1 and to 0 otherwise. Clearly, \( E[z_i] \) is equal to the original value of \( z_i \). Hence, the expected value of the constructed integer solution is the same as the cost of the optimal solution. Moreover, the constructed integer solution satisfies all the constraints of the linear program. Hence, there must exist an integer solution that is optimal.

We will show in the next section that the optimal integer solution to (5.9) can be computed in \( O(n^3) \) time (where \( n \) is the number of web services) by converting the linear program into a network flow problem [29]. Once the optimal integer solution has been found, all web services with \( z_i = 1 \) in the solution define the optimal set \( P_{C_x} \), which in turn defines the optimal cut \( C_x \).
5.4.2 Solution to Linear Program using Flows

To solve the linear program (5.9), we construct a network flow graph $W$ as follows. In $W$, let there be a node for every web service $WS_i \in \bar{H}$, a source node $s$, and a sink node $t$. Let $D = \max_{i} |WS_i \in \bar{H}| \log \sigma_i|$. For every web service $WS_i \in \bar{H}$, add a directed edge $s \rightarrow WS_i$ with capacity $D$, and a directed edge $WS_i \rightarrow t$ with capacity $D + \log \sigma_i$. Note that the choice of $D$ ensures that all capacities are non-negative. If for web services $WS_i, WS_j \in \bar{H}$, there is a precedence constraint $WS_i \prec WS_j$, add a directed edge $(WS_j, WS_i)$ with infinite capacity. For every $i$ such that $WS_i \in M_x$ (i.e., $WS_i \prec WS_x$), the capacity of the edge $(s, WS_i)$ is set to infinity.

Lemma 5.4.2. The minimum cut separating $s$ from $t$ in $W$ yields the optimal integer solution to the linear program (5.9).

Proof. Consider any feasible integer solution to program (5.9). Recall that the set of web services $WS_i \in \bar{H}$ having $z_i = 1$ define the set $P_{Cx}$. Consider the cut in $W$ defined by placing the web services in $P_{Cx}$ on the same side as $s$ and the remaining web services in $\bar{H}$ (say $P_{\bar{C}x}$) on the other side, i.e., on the same side as $t$. Due to the second constraint in (5.9), there cannot be a directed edge $(WS_i, WS_j)$ in $W$ such that $WS_i \in P_{Cx}$ and $WS_j \in P_{\bar{C}x}$. Therefore, no infinite capacity edge of the form $(WS_i, WS_j)$ crosses the cut. Since $M_x \subseteq P_{Cx}$ (by the first constraint in (5.9)), no infinite capacity edge of the form $(s, WS_i)$ crosses the cut. Therefore, the cut has finite capacity. The capacity of the cut is $\sum_{i} |WS_i \in P_{Cx}| (D + \log \sigma_i) + \sum_{i} |WS_i \in P_{\bar{C}x}| D = |\bar{H}|D + \sum_{i} |WS_i \in P_{Cx}| \log \sigma_i$ (using $|\bar{H}|$ to denote the number of web services in $\bar{H}$).

Conversely, consider any cut separating $s$ and $t$ in $W$ where the set of web services on the side of $s$ is $P_{Cx}$ and that on the side of $t$ is $P_{\bar{C}x}$. If the cut has finite capacity, the set $P_{Cx}$ must satisfy the second constraint of (5.9), and $M_x \subseteq P_{Cx}$ (the first constraint in (5.9)). The capacity of this cut is precisely $|\bar{H}|D + \sum_{i} |WS_i \in P_{Cx}| \log \sigma_i$. Therefore, the minimum cut in $W$ finds the set $P_{Cx}$ that optimizes the linear program (5.9).

The only hurdle to using an efficient network flow algorithm to compute the minimum cut in $W$ in $O(n^3)$ time (where $n$ is the number of web services) is that the capacities need not be rational numbers since they involve logarithms. In practice, this is not a problem since numbers are represented using a fixed number of bits. However, to handle the general case, we perform discretization and set $\gamma_i = \lceil n^2 \log \sigma_i \rceil$, and $\tilde{D} = \max_i |\gamma_i|$. We then
construct a modified network flow graph $W'$ exactly as $W$ except that we use $\gamma_i$ instead of $\log \sigma_i$ and $\tilde{D}$ instead of $D$. This discretization does not result in any significant loss in accuracy as shown by the following theorem.

**Theorem 5.4.3.** The minimum cut in $W'$ gives a $(1 + O(1/n))$ approximation to the optimal plan.

**Proof.** For each web service $WS_i$, we have $\log \sigma_i \leq \gamma_i/n^2 \leq \log \sigma_i + 1/n^2$. The effective selectivity of $WS_i$ after the discretization is $\sigma'_i = e^{\gamma_i/n^2}$. Thus we have $\sigma_i \leq \sigma'_i \leq \sigma_i e^{1/n^2} \leq \sigma_i(1 + 2/n^2)$. Therefore, for any set $P_{C_x}$ of web services, we have:

$$\prod_{i \mid WS_i \in P_{C_x}} \sigma_i \leq \prod_{i \mid WS_i \in P_{C_x}} \sigma'_i \leq (1 + 3/n) \prod_{i \mid WS_i \in P_{C_x}} \sigma_i$$

Thus, on using the effective selectivities $\sigma'_i$, we get a $(1 + 3/n)$ approximation to $R[P_{C_x}]$, i.e., the cost of the optimal plan. \qed

Since the capacities in $W'$ are all non-negative integers, the minimum cut in $W'$ can be found in $O(n^3)$ time [83].

### 5.4.3 Greedy Algorithm

We now describe our general greedy algorithm shown in Figure 5.6. For a partial plan DAG $\bar{H}$, we first define the frontier set $F(\bar{H})$ as the set of all web services that are candidates for addition to $\bar{H}$, since all their prerequisite web services have already been added to $\bar{H}$. Formally:

$$F(\bar{H}) = \{WS_i \mid WS_i \notin \bar{H} \land M_i \subseteq \bar{H}\} \tag{5.10}$$

We start by initializing $\bar{H}$ as empty, and the frontier set $F(\bar{H})$ as all those web services that do not have any prerequisite web services (Line 1). Then for each web service $WS_x \in F(\bar{H})$, we solve the linear program (5.9) to determine the optimal cost of adding $WS_x$ to $\bar{H}$ (Line 4). Let $WS_{opt}$ be the web service having least such cost (Line 6), and let the optimal cut for adding $WS_{opt}$ be $C_{opt}$ as given by the solution to the linear program. $WS_{opt}$ is then added to $\bar{H}$ by placing directed edges from the web services in cut $C_{opt}$ to $WS_{opt}$ (Line 7). We update the frontier set $F(\bar{H})$ according to Equation (5.10), and continue in this fashion until the DAG $\bar{H}$ includes all the web services.
Algorithm *Greedy*

1. \( \mathcal{H} \leftarrow \phi; \mathcal{F}(\mathcal{H}) \leftarrow \{ \text{WS}_i \mid M_i = \phi \} \)
2. while (\( \mathcal{H} \) does not include all web services in \( \mathcal{F} \))
3.   for each web service \( \text{WS}_x \) in \( \mathcal{F}(\mathcal{H}) \)
4.     \( v_x \leftarrow \) optimal value of linear program (5.9)
5.     \( C_x \leftarrow \) optimal cut in \( \mathcal{H} \) from the solution to (5.9)
6.     \( \text{WS}_{\text{opt}} \leftarrow \) web service \( \text{WS}_x \) with minimum \( v_x \)
7.     add \( \text{WS}_{\text{opt}} \) to \( \mathcal{H} \) placing edges from \( C_{\text{opt}} \) to \( \text{WS}_{\text{opt}} \)
8.     update \( \mathcal{F}(\mathcal{H}) \) according to Equation (5.10)

Figure 5.6: Greedy Algorithm for Bottleneck Cost Metric

### 5.4.4 Analysis of Greedy Algorithm

We now show that our algorithm *Greedy* (Figure 5.6) is correct, i.e., it produces the optimal plan. Note that since the cost of a plan is determined only by the bottleneck in the plan, in general there are many possible optimal plans. We show that our greedy algorithm finds an optimal plan. The proof is by induction on the number of web services added by *Greedy* to the partial plan \( \mathcal{H} \).

Our inductive hypothesis is that when \( k \) web services have been added to the DAG \( \mathcal{H} \) constructed by *Greedy*, \( \mathcal{H} \) agrees (in terms of edges placed) with some optimal solution restricted to just the web services in \( \mathcal{H} \), i.e., there exists an optimal solution that has \( \mathcal{H} \) as a subgraph. The base case for our induction is \( k = 0 \) which is trivially satisfied since the empty DAG is a subgraph of any DAG.

**Lemma 5.4.4.** When *Greedy* adds the \((k + 1)\)th web service, the inductive hypothesis still holds.

**Proof.** Let \( \mathcal{H} \) denote the partial DAG when \( k \) web services have been added by *Greedy*. Let \( \mathcal{H} \) be a subgraph of some optimal plan \( \mathcal{H}_{\text{opt}} \) (by the inductive assumption). Suppose the \((k + 1)\)th web service chosen by *Greedy* to be added to \( \mathcal{H} \) is \( \text{WS}_x \). Let the optimal cut in \( \mathcal{H} \) for adding \( \text{WS}_x \) be \( C_x \). An example is shown in Figure 5.7.

Consider the position of \( \text{WS}_x \) in \( \mathcal{H}_{\text{opt}} \). Suppose \( \mathcal{H}_{\text{opt}} \) has some other web service \( \text{WS}_y \in \mathcal{F}(\mathcal{H}) \) that takes input only from web services in \( \mathcal{H} \), and \( \text{WS}_x \) is placed such that \( \text{WS}_y \in P_x(\mathcal{H}_{\text{opt}}) \) (see Figure 5.7). In general, there could be many such \( \text{WS}_y \)'s that are predecessors of \( \text{WS}_x \) in \( \mathcal{H}_{\text{opt}} \); the proof remains unchanged. Modify \( \mathcal{H}_{\text{opt}} \) to \( \mathcal{H}'_{\text{opt}} \) as follows. Remove the input to \( \text{WS}_x \) in \( \mathcal{H}_{\text{opt}} \) and make its input the cut \( C_x \) (just as *Greedy* does). The output of
WS\textsubscript{x} in \( \mathcal{H}_{opt} \) is replaced by the join of the output of WS\textsubscript{x} in \( \mathcal{H}'_{opt} \) and the input to WS\textsubscript{x} in \( \mathcal{H}_{opt} \). An example of this modification is shown in Figure 5.7. We now show that \( \mathcal{H}'_{opt} \) is also an optimal plan.

**Claim 5.4.5.** On modifying \( \mathcal{H}_{opt} \) to \( \mathcal{H}'_{opt} \), the cost incurred by any web service except WS\textsubscript{x} cannot increase.

**Proof.** The only web services except WS\textsubscript{x} whose cost in \( \mathcal{H}'_{opt} \) may be different from their cost in \( \mathcal{H}_{opt} \) are those for which WS\textsubscript{x} is a predecessor in \( \mathcal{H}_{opt} \). Let \( S \) denote the set of these web services. Let \( A \) be the set \( P_x(\mathcal{H}_{opt}) \cap \mathcal{H} \) and \( B \) be the set \( P_x(\mathcal{H}'_{opt}) \). See Figure 5.7 for examples of the sets \( S, A, \) and \( B \). Note that \( B \) is the same as \( P_{C_x} \), i.e., the set that Greedy chooses to place before WS\textsubscript{x}. The combined selectivity of the set \( B - A \), i.e., \( R[B - A] \), can be at most one; if not, Greedy would have chosen \( P_{C_x} \) to be \( B \cap A \) instead of \( B \). Note that \( B \cap A \) is a feasible choice for \( P_{C_x} \) since \( A \) and \( B \) are both feasible sets of web services to place before WS\textsubscript{x}. In \( \mathcal{H}_{opt} \), the web services in \( S \) had input from the set of web services \( A \cup \{\text{WS}_{x}\} \cup \{\text{other web services} \notin \mathcal{H}\} \). In \( \mathcal{H}'_{opt} \), the web services in \( S \) have input from the expanded set of web services \( A \cup B \cup \{\text{WS}_{x}\} \cup \{\text{same other web services} \notin \mathcal{H}\} \). Since \( R[B - A] \) is at most 1, the number of data items seen by web services in \( S \) in \( \mathcal{H}'_{opt} \) is at most as many as in \( \mathcal{H}_{opt} \). Thus the cost of any web service in \( S \) cannot increase on modifying \( \mathcal{H}_{opt} \) to \( \mathcal{H}'_{opt} \). \qed

Now consider the cost incurred by WS\textsubscript{x} in \( \mathcal{H}'_{opt} \). If \( R[P_x(\mathcal{H}_{opt})] \leq R[P_x(\mathcal{H}'_{opt})] \), the cost incurred by WS\textsubscript{x} also does not increase, hence combined with Claim 5.4.5, we have \( \text{cost}(\mathcal{H}'_{opt}) \leq \text{cost}(\mathcal{H}_{opt}) \). If \( R[P_x(\mathcal{H}'_{opt})] > R[P_x(\mathcal{H}_{opt})] \), there are two cases:

1. Suppose WS\textsubscript{x} is the bottleneck in \( \mathcal{H}_{opt} \). Then the cost incurred by any other web
service, specifically by WS_y in H'_{opt}, is smaller. But then since WS_y ∈ F(\bar{H}), Greedy would have chosen WS_y to add to \bar{H} instead of WS_x. Hence this case is not possible.

2. If WS_x is not the bottleneck in H'_{opt}, then cost(H'_{opt}) is given by the cost incurred by some other web service. Hence, by Claim 5.4.5, we have cost(H'_{opt}) ≤ cost(H_{opt}).

Thus in all cases, cost(H'_{opt}) ≤ cost(H_{opt}). Since H_{opt} is an optimal plan, H'_{opt} is also optimal. After Greedy adds WS_x to \bar{H}, \bar{H} is a subgraph of H'_{opt}. Hence assuming that the inductive hypothesis holds when k web services have been added to \bar{H}, it still holds on adding the (k + 1)th web service.

\textbf{Theorem 5.4.6.} Algorithm Greedy computes an optimal plan in O(n^5) time where n is the number of web services.

\textit{Proof.} The correctness is immediate from Lemma 5.4.4 by induction on the number of web services added to \bar{H}. The running time of Greedy is at most the time taken to solve the linear program (5.9) O(n^2) times. The linear program (5.9) can be solved in O(n^3) time using a network flow algorithm [29]. Thus the total running time of Greedy is O(n^5). □

Although O(n^5) complexity may seem high, Theorem 5.4.6 is still very interesting since it demonstrates that under the bottleneck cost metric, the optimal plan can be found in polynomial time for arbitrary precedence constraints. This result is somewhat surprising given previous negative results for the analogous problem under the sum cost metric [29, 78]. Also note that the analysis in Theorem 5.4.6 to obtain the O(n^5) bound is pessimistic since it assumes the frontier set is constantly of size n; in practice, the frontier set will be smaller due to precedence constraints.

\textbf{Example 5.4.7.} We demonstrate the operation of our algorithm for optimization of the query in Example 5.2.3 with costs and selectivities as given in Example 5.2.5. Initially, WS_1 and WS_2 belong to the frontier set F(\bar{H}). Since c_1 < c_2, WS_1 is added first to the plan \bar{H}. F(\bar{H}) remains unchanged. Now to add WS_2, there are two possibilities: either after WS_1, or in parallel with WS_1. Since the former possibility has lower cost, WS_2 is added after WS_1. F(\bar{H}) is now updated to \{WS_3\}. There is only possibility for its addition: after WS_2. Thus we find that the optimal plan is a linear one as shown by Plan 1 in Figure 5.2. □
5.5 Data Chunking

There is usually some amount of overhead incurred on making any web service call, e.g., parsing SOAP/XML headers and fixed costs associated with network transmission. Hence it can be very expensive to invoke a web service separately for each tuple. To amortize the overhead, a web service may provide a mechanism to pass tuples to it in batches, or chunks. Each tuple is still treated individually by the web service, but the overall overhead is reduced.

When a chunk of input data is passed to web service $WS_i$, we assume the entire answer arrives back at the WSMS as a single chunk. The response time of $WS_i$ usually depends on the size of the input chunk. We use $r_i(k)$ to denoting the response time of $WS_i$ on a chunk of size $k$. We assume there is a limit $k_{i}^{max}$ on the maximum chunk size accepted by web service $WS_i$. Chunk-size limits can arise, e.g., from limits on network packet lengths.

The query execution algorithm with chunking is similar to that in Figure 5.3, except thread $T_i$ now reads a chunk $C$ of tuples from its input rather than a single tuple. $T_i$ extracts the set of distinct value assignments $\bar{X}$ for $X_i$ from $C$, and uses the adapter to invoke $WS_i$ with the chunk $\bar{X}$. The returned tuples are joined with the input chunk $C$, and the results are written to $T_i$’s output.

When web services can accept input in the form of chunks, the query optimizer must decide the optimal chunk size to use for each web service. The optimal chunk size for web service $WS_i$ will obviously depend on how the response time $r_i(k)$ of $WS_i$ varies as a function of the chunk size $k$. We first give an example, based on a real experiment we conducted, showing that $r_i(k)$ may depend in unexpected ways on $k$. We then show that the optimal chunk size for a web service depends only on $r_i(k)$ and is independent of the query plan in which it is being invoked, and we give an algorithm for choosing optimal chunk sizes.

Example 5.5.1. We implemented an ASP.NET web service as follows. We created a table $T(\text{int } a, \text{int } b, \text{primary key } a)$ in a commercial database system, with a clustered index on attribute $a$. The table was loaded with 100,000 tuples. The web service accepted a list of values for $a$ (the chunk) and returned the corresponding values for $b$, by issuing a SQL query to the database system in which the list of $a$ values was put in an IN clause.

We measured the response time of the web service when queried by a remote host with various chunk sizes. We found that the response time was not just linear in the chunk size, but also had a
small quadratic component to it. Thus, the time per tuple \( r(k)/k \) first decreases, and then increases with \( k \). Our current (unverified) hypothesis is that the quadratic component may be due to sorting of the \( \text{IN} \) list by the database query optimizer. The main point to glean from this example is that depending upon implementation, web service response times may vary in unexpected ways with chunk size.

The following theorem gives the optimal chunk size for each web service \( \text{WS}_i \) and shows that it is independent of the query plan.

**Theorem 5.5.2.** The optimal chunk size to be used by \( \text{WS}_i \) is \( k_i^* \) such that \( r_i(k_i^*)/k_i^* \) is minimized for \( 1 \leq k_i^* \leq k_{i \text{max}} \).

**Proof.** Let \( c_i \) denote the average response time of \( \text{WS}_i \) per input tuple as in Section 5.2.2. If \( \text{WS}_i \) uses a chunk size \( k_i \), the response time per input tuple is the total response time divided by the chunk size, i.e., \( c_i = r_i(k_i)/k_i \). Recall from Equation (5.3) that the cost of a plan is given by \( \max_{1 \leq i \leq n} \left( \prod_{j=1}^{i-1} s_j \right) c_i \). Since the selectivity values remain unchanged in the presence of chunking, the cost of the plan is minimized when \( c_i \) is minimized for each web service \( \text{WS}_i \). Hence, independent of the actual query plan, the optimal chunk size for \( \text{WS}_i \) is \( k_i^* \) such that \( r_i(k_i^*)/k_i^* \) is minimized.

In general, the response time \( r_i(k) \) of a web service \( \text{WS}_i \) may be any function of the chunk size \( k \), as demonstrated by Example 5.5.1 above. Hence, to apply Theorem 5.5.2, the optimizer relies on the Profiling and Statistics component to measure \( r_i(k) \) for different values of \( k \). Profiling may be combined with query processing by trying out various chunk sizes during query execution and measuring the corresponding response times. Once the optimal chunk size \( k_i^* \) for each web service \( \text{WS}_i \) has been determined, the optimal plan is found by setting \( c_i = r_i(k_i^*)/k_i^* \) for each \( \text{WS}_i \), and applying our query optimization algorithm from Section 5.4.

### 5.6 Implementation and Experiments

We implemented an initial prototype WSMS, described in Section 5.6.1, and performed experiments using it. Not surprisingly, in our experiments, query plan performance reflects our theoretical results (thereby validating our cost model). Using total running time of queries as a metric, we compared the plans produced by our query optimization algorithm (referred to as Optimizer) against the plans produced by the following simpler algorithms:
5.6. IMPLEMENTATION AND EXPERIMENTS

1. **Parallel**: This algorithm attempts to exploit the maximum possible parallelism by dispatching data in parallel to web services whenever possible. For example, if there are no precedence constraints, data is dispatched in parallel to all web services followed by a join at the end. An example of how this algorithm operates in the presence of precedence constraints will be given in Section 5.6.3.

2. **SelOrder**: One heuristic for efficient query processing is to reduce data as early as possible by putting the web services with lower selectivities earlier in the pipeline. **SelOrder** models this heuristic by building a (linear) plan as follows: Out of all web services whose input attributes are available, the web service with lowest selectivity is placed in the plan, and the process is repeated until all web services have been placed.

We also compared the running times of queries with and without data chunking, to demonstrate the benefits of chunking. Finally, we compared the total CPU cost at the WSMS against the cost of the slowest web service to substantiate our claim that the WSMS is not the bottleneck in pipelined processing. The main findings from our experiments are:

1. For scenarios both with and without precedence constraints, the plans produced by **Optimizer** can perform vastly better (up to about 7 times better for the problem instances we experimented with) than the plans produced by **Parallel** or **SelOrder**.

2. Using data chunking query running time can be reduced by up to a factor of 3.

3. The cost of processing at the WSMS is significantly lower than the cost of the slowest web service in the plan, demonstrating that the WSMS is not the bottleneck in a pipelined plan.

We first describe our WSMS prototype and the experimental setup in Section 5.6.1. We then describe our experiments for scenarios with no precedence constraints in Section 5.6.2, and for scenarios with precedence constraints in Section 5.6.3. In Section 5.6.4, we describe our experiments with data chunking. Finally, in Section 5.6.5, we report our results of measuring the cost incurred at the WSMS.

### 5.6.1 Prototype and Experimental Setup

The experimental setup consists of two parts: the client side, consisting of our WSMS prototype, and the server side, consisting of web services set up by us.
Our WSMS prototype is a multithreaded system written in Java. It implements Algorithm \textit{ExecutePlan} (Figure 5.3), and can execute any general execution plan with any specified chunk sizes. For communicating with web services using SOAP, our prototype uses Apache Axis [16] tools. Given a description of a web service in the Web Service Definition Language [127], Axis generates a class such that the web service can be invoked simply by calling a method of the generated class. The input and output types of the web service are also encapsulated in generated classes. Our prototype uses these classes to conveniently invoke each web service as if it were a local function call. However, since the web service that a particular thread has to interact with is known only at runtime (recall Figure 5.3), the names of the corresponding classes to be used are also known only at runtime. To get around this problem, our prototype uses Java Reflection [106] to load classes and their methods dynamically.

We use Apache Tomcat [122] as the application server and Apache Axis [16] tools for web service deployment. Each of our experimental web services \( WS_i \) runs on a different machine, and has a table \( T_i(\text{int } a, \text{int } b, \text{primary key } a) \) associated with it. \( WS_i \) is of the form \( \text{WS}_i(a^b, b^f) \): given a value for attribute \( a \), \( WS_i \) retrieves the corresponding value for attribute \( b \) from \( T_i \) (by issuing a SQL query) and returns it. Data chunking is implemented by issuing a SQL query with an IN clause. The tables \( T_i \) are stored using the lightweight IBM Cloudscape DBMS. Since attribute \( a \) is the primary key, Cloudscape automatically builds an index on \( a \). The tables \( T_i \) were each populated with tuples of the form \( (j, j) \) for \( j \) in \( \{1, \ldots, 10,000\} \).

For our experiments, we needed web services with different costs and selectivities. To obtain different costs, we introduced a delay between when a web service obtains the answer from its database and when it returns the answer to the caller of the web service. The web service cost is varied by varying this delay. Our experimental web services, as described in the previous paragraph, return exactly one tuple for each input value of attribute \( a \) (since \( a \) is a key). To obtain a selectivity \( s_i < 1 \) for web service \( WS_i \), we rejected each tuple returned by \( WS_i \) at the WSMS with probability \( 1 - s_i \). To obtain \( s_i > 1 \), for each tuple returned by \( WS_i \), we created between 1 and \( 2s_i \) new tuples, each with the same value in attribute \( a \) as the returned tuple, and randomly generated values in attribute \( b \) from the range \( 1, \ldots, 10,000 \) (so that these values of \( b \) could be used as input to another web service).

The WSMS is run on a different machine from the ones on which the web services were running. For every run, the WSMS randomly generated 2000 input tuples that formed the
input table $I$. Each input tuple had a single attribute with value in the range $1, \ldots, 10,000$. The query executed was a join of all the web services and the input table $I$. For this query, a particular execution plan $\mathcal{P}$ along with the chunk sizes to be used by each web service was specified to the WSMS. The WSMS then processed all the tuples in $I$ through the plan $\mathcal{P}$ in a pipelined fashion. Over 5 independent runs, the average processing time per tuple of $I$ is then used as a metric for comparing $\mathcal{P}$ against other plans.

### 5.6.2 No Precedence Constraints

In this experiment, we set up four web services $\text{WS}_1, \ldots, \text{WS}_4$ with no precedence constraints among them, i.e., the single attribute in the input tuples served as the input attribute to all the web services. We did not use data chunking in this experiment. With its basic functionality of one database lookup, each web service had a response time (or cost) of approximately 0.2 second per tuple. We added additional delay to control the costs of different web services.

We consider various cost ranges $\tilde{c}$, and assign $\text{WS}_1, \ldots, \text{WS}_4$ uniformly increasing costs in the range $\tilde{c}$. To ensure that different plans are produced by $\text{Optimizer}$ (which orders the web services by increasing cost according to Theorem 5.3.3), and by $\text{SelOrder}$ (which orders the web services by increasing selectivity), we assigned selectivities to web services in the reverse order of cost: the selectivities of $\text{WS}_1, \ldots, \text{WS}_4$ were set as $0.4, 0.3, 0.2, 0.1$ respectively.

Figure 5.8 shows the costs of the plans produced by the various algorithms as the range of costs $\tilde{c}$ is varied from $[0.2, 2]$ seconds per tuple to $[2, 2]$ seconds per tuple. $\text{Parallel}$ dispatches data to all web services in parallel and hence has a bottleneck cost equal to the cost of the highest-cost web service $\text{WS}_4$. $\text{SelOrder}$ puts $\text{WS}_4$ first since it has lowest selectivity, so it incurs the same bottleneck cost as $\text{Parallel}$. $\text{Optimizer}$ puts $\text{WS}_4$ first since it has lowest selectivity, so it incurs the same bottleneck cost as $\text{Parallel}$. $\text{Optimizer}$ is able to reduce the bottleneck cost to well below the cost of $\text{WS}_4$ by placing the web services in increasing order of cost. Only when all the web services become expensive does $\text{Optimizer}$ incur a cost equal to that of $\text{Parallel}$ or $\text{SelOrder}$. In this experiment we also verified that the actual per-tuple processing time is very close to that predicted by our cost model, thereby showing the accuracy of our cost model.
5.6.3 Precedence Constraints

In this experiment, we again set up for four web services $WS_1, \ldots, WS_4$. The single attribute in the input tuples served as the input attribute to $WS_1$ and $WS_2$. The output attribute from $WS_1$ (respectively $WS_2$) served as the input attribute to $WS_3$ (respectively $WS_4$). Thus, we had two precedence constraints, $WS_1 \prec WS_3$, and $WS_2 \prec WS_4$. We did not use data chunking.

$WS_1$ and $WS_2$ were set up to be proliferative, with selectivities 2 and 1 respectively. The selectivities of $WS_3$ and $WS_4$ were set as 0.1 each. The cost of each of $WS_1, \ldots, WS_3$ was set as 0.2 second per tuple. The cost of $WS_4$ was varied from 0.4 to 2 seconds per tuple.

For this scenario, Parallel chooses a plan in which data is first dispatched in parallel to $WS_1$ and $WS_2$. Then, to exploit parallelism between $WS_3$ and $WS_4$, $WS_3$ is placed in the $WS_1$ branch, and $WS_4$ in the $WS_2$ branch. Based on selectivities, SelOrder orders the web services as $WS_2, WS_4, WS_1, WS_3$. Optimizer first groups $WS_1$ and $WS_3$ together, and $WS_2$ and $WS_4$ together. Then the group containing $WS_1$ is placed before the other, since it has lower cost. Thus the overall order produced by Optimizer is $WS_1, WS_2, WS_3, WS_4$.

Figure 5.9 shows the costs of the plans produced by the various algorithms as the cost of $WS_4$ is increased. Both Parallel and SelOrder incur the cost of $WS_4$ as the bottleneck, while Optimizer reduces the bottleneck cost to below the cost of $WS_4$ by placing it last in the pipelined plan.
5.6.4 Data Chunking

In this experiment, we again set up four web services with no precedence constraints as in Section 5.6.2. For this experiment, we did not add additional delay to any web service so costs were uniform, and the selectivity of each web service was set as 0.5. The web services were arranged in a linear pipeline according to Optimizer, and the chunk size used by each web service was equal.

Figure 5.10 shows how the per-tuple cost varies as the chunk size used by the web services is increased. For comparison, we also show the per-tuple cost without chunking, i.e., with chunk size 1. Even using a chunk size of 20 reduces the per-tuple cost by more than a factor of 3. However, on increasing the chunk size further, the cost does not reduce significantly. Hence most of the benefit of chunking can be achieved even by using a relatively small chunk size.

5.6.5 WSMS Cost

In this experiment, we compared the cost incurred at the WSMS against the bottleneck web service cost in a pipelined plan. We varied the number of web services involved in the query. There were no precedence constraints, uniform web service costs, and the selectivity of each web service was set as 0.5. To demonstrate that the WSMS does not
become the bottleneck even with data chunking, each web service used a chunk size of 20.

The WSMS cost was measured with the web services arranged in a linear pipeline according to Optimizer. To demonstrate that the join threads in a plan (recall Figure 5.3) does not make the WSMS the bottleneck, we also executed another plan in which data was dispatched in parallel to all web services, and the cost of joining the results at the WSMS was measured.

Figure 5.11 shows the bottleneck cost, the WSMS cost, and the cost of the join thread as the number of web services involved in the query is increased. Even as the number of web services increases, the WSMS cost remains significantly lower than the cost of the bottleneck web service. Figure 5.11 also shows that the cost of the join thread is negligible compared to the bottleneck cost.

It is important to note that our measurements in the above experiment are only conservative, and numbers in a real setting can only be better, due to the following reasons:

- In our experiments, the WSMS and the web services were running on different machines but on the same network. Accessing web services over the internet may add an additional order of magnitude to their response time.

- Our WSMS prototype makes heavy use of the Java Reflection API [106], which is known to be extremely slow compared to direct method invocations. In a separate experiment, we found that a method call using Reflection can be 15 to 20 times slower
than a direct call to the same method. The inefficiency of Reflection is also evident in how the cost of the join thread (which does not use Reflection) compares with the rest of the WSMS cost. In the next version of our prototype, we plan to redesign the system to avoid the use of Reflection, giving up the convenience of classes generated by Axis, but decreasing the cost incurred at the WSMS by at least an order of magnitude.

Given the above factors, it is unlikely that in any real setting, the WSMS cost can become the bottleneck in the pipelined processing of a query over multiple web services.

5.7 Related Work

5.7.1 Web Service Composition

A considerable body of recent work addresses the problem of composition (or orchestration) of multiple web services to carry out a particular task, e.g. [59, 101]. In general, that work is targeted more toward workflow-oriented applications (e.g., the processing steps involved in fulfilling a purchase order), rather than applications coordinating data obtained from multiple web services via SQL-like queries, as addressed in our work. Although these approaches have recognized the benefits of pipelined processing, they have not, as far as
we are aware, included formal cost models or techniques that result in provably optimal pipelined execution strategies.

Languages such as BPEL4WS [27] are emerging for specifying web service composition in workflow-oriented scenarios. While we have not yet specifically applied our work to these languages, we note that BPEL4WS, for example, has constructs that can specify which web services must be executed in a sequence and which can be executed in parallel, similar to the presence and absence of precedence constraints in our model. We are hopeful that the optimization techniques developed here will extend to web-service workflow scenarios as they become more standardized, and doing so is an important direction for future work.

ActiveXML is a paradigm in which XML documents can have embedded web service calls in them. However, optimization work on ActiveXML [2] mostly focuses on deciding which web service calls in the document need to be made in order to answer a query posed over the XML document. As ActiveXML gains acceptance, it can be seen as an interesting mechanism to set up a distributed query plan over web services: ActiveXML fragments might be input to a web service, thereby making it invoke other web services.

5.7.2 Parallel and Distributed Query Processing

In our setting of query processing over web services, only data shipping is allowed, i.e., dispatching data to web services that process it according to their preset functionality. In traditional distributed or parallel query processing, each of which has been addressed extensively in previous work [51, 77, 103], in addition to data shipping, code shipping also is allowed, i.e., deciding which machines are to execute which code over which data. Due to lack of code shipping, techniques for parallel and distributed query optimization, e.g., fragment-replicate joins [103], are inapplicable in our scenario. Moreover, most parallel or distributed query optimization techniques are limited to a heuristic exploration of the search space whereas we provide provably optimal plans for our problem setting.

5.7.3 Data Integration and Mediators

Our WSMS architecture has some similarity to mediators in data integration systems [30, 62, 96, 107]. However, query optimization techniques for mediators, e.g., [60, 102, 130], focus mostly on issues such as choosing the right binding access pattern to access each data source,
and aim at minimizing the total consumption of resources rather than at minimizing running time by exploiting parallelism. An important focus in mediators is to optimize the cost incurred at the data integration system itself, for which classical relational database optimization techniques (or modifications thereof) often can be applied. However, our techniques focus not on optimizing the processing at the WSMS, but on optimizing the expensive web service calls by exploiting parallelism among them.

A capability frequently required in a WSMS is that of using the results from one web service to query another. This operation is essentially the same as the Dependent Join, which has been studied in [60, 93], and whose techniques are applicable in a WSMS.

5.7.4 Query Processing over Remote Sources

Exploiting parallelism among data sources has generally not been the focus of prior work. WSQ/DSQ [68] does exploit parallelism by making multiple asynchronous calls to web sources, but does not perform any cost-based optimization. Other work [80, 123] has considered adaptive query processing over remote data sources, with dynamic reoptimization when source characteristics change over time, but does not include optimizations to exploit parallelism among sources.

Our execution model of pipelined processing resembles distributed Eddies [121]. However, unlike our work, the Eddies framework does not perform static optimization of queries. A problem mathematically similar to ours has been considered in [41, 84], but only for the simpler case of no precedence constraints and all web services being selective, i.e., returning fewer data items than are input to it. Interestingly, in distributed Eddies, as well as in [41, 84], different input tuples may follow different plans, a possibility that we have not considered in our work. So far, we have focused on the problem of finding the optimal single plan for all tuples. An important direction of future work is to combine our techniques with those developed in [41, 84], thereby leading to even higher performance.

5.8 Conclusions

In this chapter, we moved on to the second new data management scenario considered in this thesis: that of data residing behind web services. We proposed the overall goal of a general-purpose Web Service Management System (WSMS), enabling clients to query a collection of web services in a transparent and integrated fashion. We focused particularly
CHAPTER 5. QUERY OPTIMIZATION OVER WEB SERVICES

on new query optimization issues that arise in a WSMS. For our execution model that consists of pipelined query processing over web services, we derived the “bottleneck” cost metric to characterize the cost of a plan. For this cost metric, we devised new algorithms to: (a) decide the optimal arrangement of web services in a pipelined plan, respecting precedence constraints, and (b) decide the optimal chunk size to use when sending data to each web service. Finally, we provided an experimental evaluation of our optimization algorithms using an initial WSMS prototype we developed. Our experiments demonstrated that both with and without precedence constraints, the plans produced by our algorithms can perform vastly better than those produced by more naïve algorithms, and that the plan performance can be further improved by using data chunking. Also, as assumed in the derivation of the bottleneck cost metric, our experiments demonstrated that the WSMS is not expected to be the bottleneck in pipelined plans.
Chapter 6

Building Histograms Without Data Access

In the last chapter, in which we investigated optimization of queries over web services, we assumed that certain knowledge about the data distribution was available. For example, the selectivity of a filter depends on the data items to which the filter is applied. Usually, selectivity estimates are made by keeping statistics: a summary of the underlying data distribution. In this chapter, we focus on histogram summaries, which are one of the most common statistics to be kept for query optimization. We consider the problem of how these statistics are gathered, i.e., how histograms are built. As described in Section 1.2.2, the key challenge in gathering statistics in the web services scenario is that web services usually do not allow direct access to the data they manage, which is required for typical statistics collection. Even in conventional databases, where direct data access is available, gathering statistics by accessing the data is not always a good approach, as will be discussed in Section 6.1. Thus, we address the general problem of building histograms without direct data access, not tied to a specific environment such as web services.

In the absence of direct data access, the only way that information may be gained about the underlying data distribution is by observing the results of queries. In this chapter, we describe a new technique for building histograms by observing query results. Our techniques were developed and implemented in the context of the conventional IBM DB2 database system [111]. Although the same ideas carry over for the web services scenario, there are a number of research issues that are left to be addressed before this work can become a complete solution for the profiling component of a WSMS (recall Section 5.1).
We discuss these issues in Section 6.9.

6.1 Introduction

Query planning typically relies heavily on data statistics. For example, we saw in Chapters 2, 4, and 5 that the selectivity of filters occurring in queries was critical to selecting the best query plans. In Chapter 3, our techniques relied on the average number of join tuples produced by every input tuple, which is determined by the selectivity of the join predicate. Selectivity estimates are usually made by keeping statistics that are essentially summaries of the underlying data distribution. There are various types of statistics or summaries, e.g., histograms [85], wavelets [95], and sketches [8]. We focus on histograms, the most common type of summary.

Histograms approximate the data distribution by dividing the data range into buckets and keeping a count of the number of data items in each bucket. For example, in Example 5.1.1, the selectivity of the predicate ‘cr > threshold’ can be estimated by keeping a histogram over the data values for the credit rating (cr) attribute. The more closely the histogram reflects the actual data distribution, the more accurate it is.

Queries may have filters on more than one attribute: in Example 5.2.3 there are two filters, one over credit rating, and another over payment history. While the selectivity of each of these single-attribute filters can be estimated well by using a histogram on the corresponding attribute, it can be problematic to estimate the combined selectivity of these filters. In Chapter 5, as a first step, we assumed filters were independent. Thus, the combined selectivity of two filters, i.e., the chance that a data item satisfies both filters, is estimated as the product of the selectivities of the two single-attribute filters. In reality however, attributes may be correlated, leading to high errors in selectivity estimation, and hence in query planning. For example, credit rating may be highly correlated with payment history, since people having a good payment history usually have a high credit rating. To accurately estimate the combined selectivity of multiple filters involving different attributes, multidimensional histograms [99, 105] have been proposed. Such a histogram captures the joint data distribution over the attributes on which it is constructed. These attributes need not necessarily be from the same table\(^1\). Thus, for Example 5.2.3, it may be beneficial to

\(^1\)If the attributes are from different tables, the histogram is constructed over a view that is a natural join of the tables containing these attributes.
construct a 2-dimensional histogram on the attributes credit rating and payment history.

Let us now consider the question of how histograms are built. The traditional method of building histograms in a DBMS is through a data scan (henceforth called the proactive method). The proactive method has some disadvantages. First, it does not scale well to large tables: scans can become prohibitive with large data sizes. Second, after a histogram has been built proactively, if the data is updated, the histogram becomes out-of-date and requires a rescan of the data to be able to provide up-to-date estimates. Finally, the proactive method does not use information about the workload: the set of queries that are typically posed to the DBMS. Consequently, the proactive method can waste resources trying to accurately capture parts of the data distribution that are not going to be queried at all. In our running example, capturing the credit rating distribution among people having a bad payment history is not useful. For a WSMS, the proactive method is simply infeasible because direct access to the data behind web services is usually not possible.

A practical alternative [3, 28, 37, 87] for building histograms that addresses the above problems is to gather information during query execution about the observed selectivities of filters in the queries actually executed, and then use this query feedback to build a histogram (henceforth called the reactive method). For example, suppose that we are building a histogram for the make attribute of a Car table. Consider a query having a filter make = 'Honda', and suppose that the execution engine finds at runtime that 80 tuples from the Car table satisfy this filter. Such a piece of information about the number of items satisfying a filter is called a query-feedback record (QFR). As the DBMS executes queries, QFRs can be collected with relatively little overhead [113] and used to build and progressively refine a histogram over time. For example, the above QFR may be used to refine a histogram on make by creating a bucket for 'Honda', and setting its count to 80. We refer to this process as “adding a QFR to the histogram”.

The reactive method is attractive for several reasons. Its performance does not degrade with large table sizes: it does not matter whether there are 80 Honda cars or 80000; the histogram maintenance cost is exactly the same. Since the reactive method does not access the data, it does not contend with query execution. A reactively-built histogram requires no periodic rebuilding of the histogram because updates can be incorporated automatically by new QFRs. The histogram buckets may be chosen to best suit the current workload, thereby efficiently focusing resources on precisely those queries that matter most to the user. For example, if Toyota cars are never queried, there will not be a bucket to capture
the number of Toyota cars.

Challenges

Realizing a reactive approach to histogram maintenance is a difficult problem, and previous proposals have lacked either in accuracy or efficiency. Some proposals, e.g., STGrid [3], use heuristics to add a new QFR to the histogram, thereby leading to inaccuracies in the constructed histogram. Other proposals, e.g., STHoles [28], require extremely detailed feedback from the query execution engine that can be very expensive to gather at runtime. These inadequacies of earlier approaches are illustrated through a running example in Section 6.3.2. Achieving accuracy and efficiency at the same time entails a number of interesting challenges:

1. **Enforcing consistency**: To ensure gain in histogram accuracy over time, when information is obtained from new QFRs, the histogram should not lose consistency with the information obtained from previous QFRs (unless previous QFRs have been invalidated due to data updates). Previous proposals for feedback-driven histogram construction [3, 87] have lacked this crucial consistency property, even when the data is not updated.

2. **Dealing with data changes**: In the presence of updates, deletes, and inserts, some of the QFRs collected in the past may no longer be valid. Such old, invalid QFRs must be efficiently identified and discarded, and their effect on the histogram must be undone.

3. **Meeting a limited space budget**: Since the histogram is meant to be a summary, there may be a limit on its size (perhaps a few disk pages). This size limit is imposed to ensure efficiency when the histogram is read at the time of query optimization. As more QFRs are added to the histogram while maintaining consistency, there is an increase in histogram size. To keep the histogram within the space budget, the relatively “important” QFRs (those that refine parts of the histogram that are not refined by other QFRs) must be identified and retained, and the less important QFRs discarded.

In this chapter, we describe ISOMER (Improved Statistics and Optimization by Maximum-Entropy Refinement), a new algorithm for reactive histogram construction. In contrast to
previous approaches, ISOMER is both accurate as well as efficient. ISOMER addresses each of the above challenges using novel, efficient techniques:

1. ISOMER uses the **maximum-entropy principle** (Section 6.4.1) to approximate the true data distribution by the “simplest” distribution that is consistent with all of the currently valid QFRs. This approach amounts to imposing uniformity assumptions (as made by traditional optimizers) when, and only when, no other statistical information is available. In this manner, ISOMER avoids incorporating extraneous—and potentially erroneous—assumptions into the histogram. ISOMER efficiently updates the maximum-entropy approximation in an incremental manner as new QFRs arrive.

2. ISOMER employs linear programming to quickly detect and discard old, invalid QFRs.

3. An elegant feature of ISOMER is that the maximum-entropy solution automatically yields an “importance” measure for each QFR. Thus, to meet a limited space budget, ISOMER simply needs to discard QFRs in increasing order of this importance measure.

We have conducted experiments (Section 6.8) that indicate that for reactive histogram maintenance, ISOMER is highly efficient, imposes very little overhead during query execution, and can build much more accurate histograms than previous techniques.

**Chapter Organization**

The rest of the chapter is organized as follows. We lay out the basic architecture of ISOMER in Section 6.2 and introduce its various components. In Section 6.3, we introduce notation and give a running example that illustrates the challenges in developing a reactive approach, the shortcomings of the previous approaches, and how our approach addresses these challenges. In Section 6.4, we describe how ISOMER uses the maximum-entropy principle to build histograms consistent with query feedback. Section 6.5 describes how ISOMER deals with changing data, and Section 6.6 details how ISOMER keeps the histogram within a limited space budget by discarding relatively unimportant QFRs. We summarize ISOMER in Section 6.7 and then describe our experimental results in Section 6.8. Finally, we survey related work in Section 6.10, and conclude in Section 6.11.
6.2 ISOMER Architecture and Overview

Figure 6.1 shows the architecture of ISOMER in the upper right and how it connects with a data management system. The left portion is similar to our original Figure 1.1, and represents a data management system that could be either a traditional DBMS, a data stream system as in Chapters 2-4, or a WSMS as in Chapter 5. The proactive approach for maintaining statistics that requires database access is shown by the dashed arrow.

Now consider statistics collection using ISOMER. QFRs are collected in a query feedback store, then used periodically to build and refine the histogram. In principle, the use of a query feedback store is not necessary: we could instead use each QFR to refine the histogram as soon as it is obtained. However, due to the presence of the query feedback store, the system can choose to invoke ISOMER only during periods of light load, or during a maintenance window, so that query processing performance won’t be adversely affected. Also, the batching of QFRs is much more efficient since the cost of invoking ISOMER is amortized.

The right portion of the figure shows the various components of ISOMER. When invoked, ISOMER reads the currently maintained histogram on disk, accesses the new QFRs, and adds them to the current histogram. This process mainly involves forming buckets corresponding to the new QFRs, and is explained in Section 6.3.3.
Enforcing consistency with previously added QFRs is one of the goals of ISOMER (Section 6.1). However, because of updates to the data these QFRs may be contradictory, in which case there does not exist a histogram consistent with all QFRs. Thus ISOMER’s next task is to detect and discard old, inconsistent QFRs. For this purpose, ISOMER keeps a list of QFRs currently incorporated in the histogram in an offline store as depicted in Figure 6.1. ISOMER reads the previous QFRs from the offline store and uses linear programming to find and eliminate inconsistent QFRs. The size of the offline store is not a concern since it cannot grow bigger than the size of the maintained histogram: if there are more QFRs than buckets, then some of these QFRs will be either inconsistent or redundant and hence will be eliminated.

Once ISOMER obtains a consistent set of QFRs, ISOMER incrementally recomputes the new histogram from the previous one according to the maximum-entropy principle. We describe this computation in Section 6.4. If the histogram is too large to fit within its allotted space, ISOMER selectively discards the relatively “unimportant” QFRs in order to reduce the histogram size. Intuitively, a particular QFR is unimportant if the information provided by that QFR is already provided by other QFRs, i.e., if the QFR refines a portion of the histogram that is already sufficiently refined by other QFRs. The process for detecting and discarding unimportant QFRs is described in Section 6.6. Based on which feedback are discarded, the final histogram is again incrementally recomputed according to the maximum-entropy principle and written back to disk.

6.3 Preliminaries

6.3.1 Notation

Given a table $T$ comprised of $N$ tuples, we wish to build a $d$-dimensional histogram over attributes $A_1, \ldots, A_d$ of table $T$. For each numerical attribute $A_i$, denote by $l_i$ and $u_i$ the minimum and maximum values of $A_i$ in the table. In the case of IBM DB2, these values are available from one-dimensional database statistics on $A_i$. A categorical attribute $A_i$ having $D_i$ distinct values can be treated as numerical by mapping each distinct value to a unique integer in $[1, D_i]$, so that $l_i = 1$ and $u_i = D_i$. The space in which the tuple values lie is given by $S = [l_1, u_1] \times [l_2, u_2] \times \ldots \times [l_d, u_d]$. This knowledge of $S$ is not available in the case of a WSMS; the resulting issues are discussed in Section 6.9.
A multidimensional histogram is a lossy compressed representation of the true distribution of tuples in \( S \) that is obtained by partitioning \( S \) into \( k \geq 1 \) non-overlapping regions called \textit{buckets}, and recording the number of tuples in \( T \) that fall in each bucket. ISOMER actually maintains an approximate histogram that records an estimate of the number of tuples that fall in each bucket. Denote by \( b_1, b_2, \ldots, b_k \) the \( k \) histogram buckets, by \( C(b_i) \) the region of \( S \) that is covered by \( b_i \), and by \( n(b_i) \) the number of tuples estimated to lie in \( b_i \). The tuples in each bucket \( b_i \) are assumed to be uniformly distributed throughout \( C(b_i) \).

ISOMER maintains a histogram based on query feedback. Specifically, at each time point, the system maintains a list of QFRs of the form \((q_1, N(q_1)), (q_2, N(q_2)), \ldots, (q_m, N(q_m))\) for some \( m \geq 1 \), where each \( q \) is a filter of the form
\[
(x_1 \leq C_1 \leq y_1) \land \cdots \land (x_{d'} \leq C_{d'} \leq y_{d'})
\]
and \( N(q) \) is the number of tuples that satisfy \( q \). Here \( d' \leq d \) and \( C_1, \ldots, C_{d'} \) are distinct attributes from among \( A_1, \ldots, A_d \). We assume that if \( C_j \) is categorical, then \( x_j = y_j \), so that the conjunct \( x_j \leq C_j \leq y_j \) is actually an equality filter. We denote by \( R(q) \) the subset of the region \( S \) for which \( q \) is true.

The maximum-entropy principle used by ISOMER is general enough to apply to any kind of filters and not just to conjunctive filters of the form above. However, the histogram structure currently used by ISOMER, described in Section 6.3.3, supports only conjunctive filters.

### 6.3.2 A Running Example
Consider a Car relation with attributes make and color, and suppose that a user executes the query
\[
\text{SELECT * FROM Car WHERE make = 'Honda' AND color = 'White'}
\]
Figure 6.2 shows a possible plan for executing this query; the actual number of tuples at each stage are shown in parentheses. The figure also shows the various filters \( q \) for which \( N(q) \) can be collected during query execution; the cardinality information for these filters comprise the QFRs. Gathering of QFRs is already supported by several commercial database systems and has been shown [113] to have very low runtime overhead (typically less than 5%).

Suppose that we wish to build a two-dimensional histogram on the attributes make and color by using query feedback. Also suppose for simplicity that there are only two distinct makes (say 'Honda' and 'BMW'), and only two distinct colors (say 'Black' and 'White') in the Car relation. Then any multidimensional histogram on these attributes has at most four buckets (Figure 6.3), and suppose that all of these four buckets are stored in our histogram.

Suppose we know (e.g., from system catalog statistics) that the total number of tuples in the Car table is 100. Again, such information may not be available in the case of a WSMS (see Section 6.9). This QFR can be expressed as:

\[
  n(b_1) + n(b_2) + n(b_3) + n(b_4) = 100 \tag{6.1}
\]

At this point, there are various possible assignments of values to the \( n(b_i) \)'s that will make the histogram consistent with this single QFR. In the absence of any additional knowledge, we assume that values are distributed uniformly. Hence, the histogram obtained after adding this QFR is one in which each \( n(b_i) \) equals 25, as shown in Figure 6.3(b).

Suppose we next add the QFR \( N(\text{make} = \text{'Honda'}) = 80 \) to the histogram. This QFR can be expressed as:

\[
  n(b_2) + n(b_4) = 80 \tag{6.2}
\]

To make the histogram consistent with (6.2) while preserving uniformity between \( n(b_2) \) and \( n(b_4) \), we set \( n(b_2) = n(b_4) = 40 \). The resulting histogram is shown in Figure 6.3(c). At this point, the STGrid [3] approach would consider the process of adding this QFR to be finished, with Figure 6.3(c) as the final histogram.

Notice, however, that the histogram in Figure 6.3(c) is no longer consistent with (6.1). The expressions in (6.1) and (6.2) together imply that \( n(b_1) + n(b_3) = 20 \). To enforce consistency with this equation as well as uniformity between \( n(b_1) \) and \( n(b_3) \), we set \( n(b_1) = n(b_3) = 10 \). The final consistent histogram is shown in Figure 6.3(d).
Observe that even though we added information only about Hondas, the histogram now gives a more accurate estimate of the frequency of BMWs. This improvement is a direct result of our final histogram adjustment in which we enforced consistency with both the QFRs.

In summary, we impose the following requirements when adding a new QFR to the histogram:

1. **Consistency**: After adding a new QFR, the histogram should be consistent with all QFRs added so far.

2. **Uniformity**: If there are multiple histograms consistent with all of the QFRs, then the choice of the final histogram should not be arbitrary, but must be based on the traditional assumption of uniformity.

So far in this example, it has been easy and intuitive to apply the uniformity assumption. However, it is not always clear how to impose uniformity. To illustrate, suppose that the next QFR we add to the histogram is \(N(\text{color} = '\text{white}') = 30\). This QFR can be written as:

\[
n(b_3) + n(b_4) = 30. \tag{6.3}
\]

If we employ the naïve solution of making the histogram consistent with (6.3) while imposing uniformity on \(n(b_3)\) and \(n(b_4)\), we get \(n(b_3) = n(b_4) = 15\). Further enforcing consistency with (6.1) and (6.2), we get the final histogram shown in Figure 6.3(e). This solution clearly does not enforce uniformity\(^2\): although white Hondas and white BMWs are in equal proportion, there are far more black Hondas than black BMWs. This non-uniformity among black cars has not been indicated by any added QFR, and is only due to our ad-hoc method of enforcing consistency.

\(^2\text{STHoles [28] does not face this problem of enforcing uniformity because for a filter } q, \text{ it explicitly gathers a count of the intersection of } R(q) \text{ with every histogram bucket. Thus, STHoles would gather } n(b_3) \text{ and } n(b_4) \text{ individually, making it a high-overhead approach in general.}\)
Clearly, we need a more principled way of adding QFRs to the histogram. Before describing how ISOMER tackles this problem (Section 6.4), we describe the histogram structure used by ISOMER (Section 6.3.3).

6.3.3 Histogram Structure in ISOMER

ISOMER uses the STHoles data structure \[28\] to represent and store multidimensional histograms. In the STHoles histogram, each bucket \( b_i \) has a hyperrectangular bounding box denoted by \( \text{box}(b_i) \subseteq S \), i.e., bucket \( b_i \) is bounded between two constant values in each dimension. Bucket \( b_i \), however, does not cover the entire region \( \text{box}(b_i) \). There may be some “holes” inside \( \text{box}(b_i) \) that are not covered by \( b_i \). These regions are themselves histogram buckets, and are referred to as children of \( b_i \). The bounding boxes of these children are mutually disjoint hyperrectangles and completely enclosed within \( \text{box}(b_i) \). The region covered by \( b_i \) is formally given by:

\[
C(b_i) = \text{box}(b_i) - \bigcup_{b_j \in \text{children}(b_i)} \text{box}(b_j)
\]

Intuitively, in the absence of holes, \( b_i \) would represent a region of uniform tuple density. However, the STHoles histogram identifies regions within \( b_i \) that have a different tuple density, and represents them as separate histogram buckets.

For a multidimensional filter \( q \) that selects all tuples lying in a specified region \( R(q) \subseteq S \), an STHoles histogram comprising buckets \( b_1, b_2, \ldots, b_k \) estimates the number of tuples that satisfy this filter as

\[
\hat{N}(q) = \sum_{i=1}^{k} \frac{n(b_i)}{\text{vol}(C(b_i))} \text{vol}(R(q) \cap C(b_i)).
\]

Here \( \text{vol}(R) \) denotes the usual euclidean volume of the region \( R \) when the data is real-valued; for discrete (i.e., integer or integer-coded categorical) data, \( \text{vol}(R) \) denotes the number of integer points that lie in \( R \).

ISOMER initializes the STHoles histogram to contain a single bucket \( b_1 \) such that \( \text{box}(b_1) = C(b_1) = S \) and \( n(b_1) = N \). At this point, the histogram embodies the simplest possible uniformity assumption. As more QFRs are added over time, ISOMER learns more about the distribution of tuples in \( S \) and incorporates this information into the histogram.
by “drilling” holes in $b_1$.

ISOMER’s technique for drilling holes is a simpler version of the method given in [28]. Suppose that ISOMER obtains a QFR about a specified multidimensional filter $q$. To make the histogram consistent with this QFR, ISOMER must first ensure that the histogram contains a set of buckets that exactly cover $R(q)$, so that the sum of the tuple counts in these buckets can then be equated to $N(q)$ as in Section 6.3.2. If such a set of buckets already exists, no holes need to be drilled. Otherwise, the process of drilling holes for $q$ proceeds as shown in Figure 6.4. Specifically, ISOMER descends down the bucket tree until it finds a bucket $b$ such that $R(q) \subset C(b)$ but $R(q) \nsubset C(b')$ for any $b' \in \text{children}(b)$. ISOMER forms a new bucket $b_{\text{new}}$ such that box($b_{\text{new}}$) = $R(q)$ and processes each bucket $b' \in \text{children}(b)$ as follows.

- If box($b'$) $\cap$ $R(q)$ = $\emptyset$, then nothing needs to be done.
- If box($b'$) $\subset$ $R(q)$, then $b'$ is removed from children($b$) and added to children($b_{\text{new}}$).
- If box($b'$) partially overlaps $R(q)$, then bucket $b'$ (and recursively its children), are split as shown in Figure 6.4 to preserve disjointness and a hyperrectangular shape. The splitting is done one dimension at a time, using an arbitrary ordering among the dimensions.

### 6.4 Incorporating Feedback

To meet the dual goals of consistency and uniformity (as described in Section 6.3.2), and to avoid unsatisfactory results as in Figure 6.3(e), ISOMER turns to the information-theoretic principle of maximum entropy, described next.
6.4. INCORPORATING FEEDBACK

6.4.1 The Maximum-Entropy Principle

Consider a discrete probability distribution \((p_1, p_2, \ldots, p_n)\) on \(n\) distinct outcomes, i.e., each \(p_i\) is nonnegative and \(\sum_{i=1}^{n} p_i = 1\). In many applications, only partial information about such a probability distribution is available (e.g., \(p_1 + p_2 = 0.5\)). Define a “candidate” distribution as one that is consistent with all available information about the distribution. In general, there may be multiple candidate distributions. The maximum-entropy principle (see, e.g., [108]) provides a well-grounded criterion for selecting a unique distribution from among the candidates. Specifically, the principle prescribes selection of the candidate \(P = (p_1, p_2, \ldots, p_n)\) that has the maximum entropy value \(H(P)\), where \(H(P) = -\sum_{i=1}^{n} p_i \ln(p_i)\). The maximum-entropy principle can be justified informally as follows. From information theory, we know that entropy measures the uncertainty or un informativeness in a distribution. For example, the value of the entropy ranges from 0—when a specified outcome occurs with certainty—to a maximum of \(\ln(n)\) when no information is available and all outcomes are equally likely \((p_1 = \cdots = p_n = 1/n)\). Thus the maximum-entropy principle leads to a choice of the simplest, i.e., most uninformative distribution possible that is consistent with the available information. To choose a distribution with lower entropy would amount to assuming information that we do not have; to choose one with a higher entropy would ignore the information that we do have. The maximum-entropy distribution is therefore the only reasonable choice. A more formal justification of the principle can be found in [108]. For a continuous probability distribution with probability density function (pdf) \(P(u)\), the entropy is defined as \(H(P) = \int_{S} P(u) \ln(P(u)) \, du\), and the foregoing discussion extends to this setting directly.

For ISOMER, the maximum-entropy principle can also be justified as a means of ensuring uniformity. As mentioned above, entropy is maximized for a uniform distribution. Thus, choosing the distribution according to the maximum-entropy principle facilitates our goal of maintaining consistency and uniformity at the same time.

6.4.2 Formulation of the Optimization Problem

To apply the maximum-entropy principle in ISOMER, we associate a probability distribution \(P\), and hence an entropy value \(H(P)\), with every possible histogram. In accordance with the maximum-entropy principle, ISOMER then maximizes \(H(P)\) over the set of all histograms that are consistent with the current set of QFRs. To define \(P\) and \(H(P)\), consider an STHoles histogram with buckets \(b_1, \ldots, b_k\) having bucket counts \(n(b_1), \ldots, n(b_k)\).
If the data is discrete, then \( P \) is a probability distribution over the integer points of \( S \) given by 
\[
p_u = \frac{n(b_u^*)}{N \cdot V(b_u^*)}
\]
for \( u \in S \), where \( V(b) \) is abbreviated notation for \( \text{vol}(C(b)) \), and \( b_u^* \) is the unique bucket \( b \) such that \( u \in C(b) \). This definition follows from (6.4) after dividing both sides by the total number of tuples \( N \) and taking \( q \) to be the point query \( "(A_1, A_2, \ldots, A_d) = u" \). The entropy \( H(P) = -\sum_{u \in S} p_u \ln(p_u) \) corresponding to the distribution \( P \) is thus given by

\[
H(P) = -\sum_{u \in S} \frac{n(b_u^*)}{N \cdot V(b_u^*)} \ln \left( \frac{n(b_u^*)}{N \cdot V(b_u^*)} \right)
\]

Since the inner sum comprises \( V(b_i) \) identical terms that are independent of \( u \),

\[
H(P) = -\sum_{i=1}^{k} \frac{n(b_i)}{N} \ln \left( \frac{n(b_i)}{N \cdot V(b_i)} \right)
\]

where the last equality uses the identity \( \sum_{i=1}^{k} n(b_i) = N \). For real-valued data, we take \( P \) to be the pdf defined by \( P(u) = p_u \) for each real-valued point \( u \in S \), where \( p_u \) is defined as above; note that this density function is constant within each region \( C(b_i) \). A straightforward calculation shows that the entropy \( H(P) = \int_S P(u) \ln(P(u)) \, du \) is given by (6.5).

We now express the QFRs as constraints on the histogram. Suppose that ISOMER has obtained QFRs for \( m \) filters \( q_1, \ldots, q_m \). First, ISOMER drills holes for these QFRs in the histogram as described in Section 6.3.3. For each \( q_i \), the drilling procedure ensures that the set of histogram buckets lying within \( R(q_i) \) exactly cover \( R(q_i) \). Hence the QFR for \( q_i \) can be written as the constraint

\[
\sum_{b \in C(b) \subseteq R(q_i)} n(b) = N(q_i)
\]
The application of the maximum-entropy principle thus leads to a well posed\(^3\) optimization problem: for an STHoles histogram with buckets \( b_1, \ldots, b_k \), select nonnegative bucket counts \( n(b_1), \ldots, n(b_k) \) so as to maximize the expression \( H(P) \) in (6.5), while satisfying (6.6) for \( 1 \leq i \leq m \).

### 6.4.3 Solution of Optimization Problem

To solve the above optimization problem, associate a Lagrange multiplier \( y_i \) (\( 1 \leq i \leq m \)) with the \( i \)th constraint given by (6.6). After removing the constants from the objective function in (6.5), the Lagrangian of the optimization problem is given by:

\[
\sum_{i=1}^{m} y_i \left( \sum_{b \in C(b) \subseteq R(q_i)} n(b) - N(q_i) \right) - \sum_{i=1}^{k} n(b_i) \ln \left( \frac{n(b_i)}{V(b_i)} \right)
\]

Differentiate the above expression with respect to \( n(b) \) and equate to 0 to get

\[
\sum_{i \in C(b) \subseteq R(q_i)} y_i - \ln \left( \frac{n(b)}{V(b)} \right) - 1 = 0,
\]

so that, setting \( \lambda_i = e^{y_i} \),

\[
n(b) = \frac{V(b)}{e} \prod_{i \in C(b) \subseteq R(q_i)} \lambda_i.
\]  

(6.7)

Combining (6.7) with (6.6), we get a system of \( m \) equations, where there are \( m \) unknowns \( \lambda_1, \ldots, \lambda_m \). Substituting (6.7) into (6.6), we obtain

\[
\sum_{b \in C(b) \subseteq R(q_i)} \left( \frac{V(b)}{e} \prod_{j \in C(b) \subseteq R(q_j)} \lambda_j \right) = N(q_i),
\]

\[
\Rightarrow \quad \lambda_i = \frac{N(q_i) \cdot e}{\sum_{b \in C(b) \subseteq R(q_i)} \left( V(b) \prod_{j \in C(b) \subseteq R(q_j)} \lambda_j \right)}
\]  

(6.8)

for \( 1 \leq i \leq m \). We refer to (6.8) as the *iterative scaling equation*. We can start with any convenient initial value for the \( \lambda_i \)'s (e.g., \( \lambda_1 = \cdots = \lambda_m = 1 \)) and then use (6.8) to iteratively update the value of each \( \lambda_i \) in turn until convergence.

\(^3\)There may be some buckets \( b \) for which \( n(b) = 0 \) in any consistent solution. We detect such buckets by solving an appropriate linear program, and exclude them from the entropy calculation since their contribution to the entropy is 0 (\( \lim_{n(b) \rightarrow 0} n(b) \ln (n(b)) = 0 \)).
Example 6.4.1. For the example of Section 6.3.2, \( V(b_i) = 1 \) for each \( i \). As before, we add three QFRs given by (6.1), (6.2), and (6.3). Denote these QFRs as \( q_1, q_2, \) and \( q_3 \), respectively; see Figure 6.5. Since bucket \( b_1 \) is referred to only by \( q_1 \), it follows from (6.7) that \( n(b_1) = \lambda_1/e \). Similarly, \( n(b_2) = \lambda_1\lambda_2/e, n(b_3) = \lambda_1\lambda_3/e, \) and \( n(b_4) = \lambda_1\lambda_2\lambda_3/e \). We can therefore rewrite (6.1), (6.2), and (6.3) as

\[
\lambda_1 + \lambda_1\lambda_2 + \lambda_1\lambda_3 + \lambda_1\lambda_2\lambda_3 = 100e \\
\lambda_1\lambda_2 + \lambda_1\lambda_2 = 80e \\
\lambda_1\lambda_3 + \lambda_1\lambda_2\lambda_3 = 30e
\]

Solving the above equations, we obtain \( \lambda_1 = 14e \), \( \lambda_2 = 4 \), and \( \lambda_3 = 3/7 \), yielding the final histogram (Figure 6.5).

This histogram is consistent with all of the added QFRs. It is also “the most uniform” in the following sense. It maintains the 80-20 ratio between Hondas and BMWs for both the colors. Similarly, it maintains the 30-70 ratio between white and black cars for both the makes. Such uniformity is not obtained by adding QFRs in an ad-hoc manner, e.g., as in the histogram of Figure 6.3(e).

Our implementation of ISOMER uses two techniques to speed up iterative scaling.

1. **Incremental computation**: When a new QFR is added, the new maximum-entropy solution can be computed incrementally using the previous solution. To enable incremental computation, ISOMER persists the multipliers corresponding to each QFR in the offline store (Section 6.2). When the next QFR (for filter \( q_{m+1} \)) is added, the old values of \( \lambda_1, \ldots, \lambda_m \) together with \( \lambda_{m+1} = 1 \) form the starting point for the iterative scaling. In this way, the solution needs to be recomputed only for \( R(q_{m+1}) \) and those regions
covered by QFRs that intersect with $R(q_{m+1})$. For the rest of the histogram space $S$, the previous solution is reused. In a similar manner, ISOMER computes a new solution incrementally whenever a constraint is discarded, using the old multiplier values as the starting point.

2. **Caching of products**: For each bucket $b$, we keep the precomputed product $\prod_{j | C(b) \subseteq R(q_j)} \lambda_j$. This product is referred to as the bucket multiplier corresponding to bucket $b$. With bucket multipliers available, we avoid the need to repeatedly compute the product in the denominator on the right side of (6.8). When a particular $\lambda_i$ is updated according to (6.8), ISOMER updates the multipliers corresponding to all buckets $b$ such that $C(b) \subseteq R(q_i)$. Note that ISOMER must iterate through all such buckets in any case, in order to compute the sum in the denominator of the right side of (6.8). Thus bucket multipliers can be maintained at low cost and lead to substantial savings in computation.

### 6.5 Dealing with Database Updates

As long as no database tuples are updated, inserted, or deleted, all of the QFRs obtained by ISOMER are consistent with each other, and there exists at least one valid histogram solution that satisfies the set of constraints in (6.6). However, in the presence of data changes, the set of QFRs might evolve to a point at which no histogram can be simultaneously consistent with the entire set.

**Example 6.5.1.** Suppose that ISOMER obtains the two QFRs $N(\text{make} = \text{'Honda'}) = 80$ and $N(\text{make} = \text{'Honda'}, \text{color} = \text{'White'}) = 30$, and then some updates to the data occur. After these updates, ISOMER might obtain the QFR $N(\text{make} = \text{'Honda'}, \text{color} = \text{'Black'}) = 60$. Clearly, there exists no histogram solution consistent with all three QFRs.

Given inconsistent QFRs, there exists no solution to the optimization problem of Section 6.4.2. Thus, ISOMER must first discard the QFRs that are no longer valid due to data changes, leaving a set of consistent QFRs having a valid histogram solution. However, deciding which QFR is invalid is not always straightforward and depends on the type of data change. In Example 6.5.1, if some new black-Honda tuples have been inserted, the first QFR is invalid. However, if the color of some Hondas has been updated from white to black, then the second QFR is invalid. In general, both the first and second QFRs may be invalid.
CHAPTER 6. BUILDING HISTOGRAMS WITHOUT DATA ACCESS

Since no information is available about the type of data change, ISOMER uses the notion of the age of a QFR to decide which QFRs to discard. The intuition is that the older a QFR, the more likely that it has been invalidated. Thus ISOMER discards those QFRs that are relatively old and whose removal leaves behind a consistent set. To quickly detect such QFRs, the following LP approach is employed.

6.5.1 A Linear-Programming Solution

ISOMER associates two “slack” variables with each constraint corresponding to the QFRs. The constraints in (6.6) are rewritten as (for $1 \leq i \leq m$)

$$\sum_{b \in C(b) \subseteq R(q_i)} n(b) - N(q_i) = s_i^+ - s_i^-$$

(6.9)

ISOMER also adds the nonnegativity constraints

$$n(b) \geq 0 \text{ for all } b, \quad s_i^+, s_i^- \geq 0 \text{ for } i = 1, \ldots, m.$$  

(6.10)

If there is a solution to the set of constraints (6.9) and (6.10) such that $s_i^+ = s_i^- = 0$, then the solution satisfies the $i$th constraint from (6.6). Otherwise, if $s_i^+$ or $s_i^-$ is positive, the $i$th constraint is not satisfied. Ideally, we would like a solution that satisfies the maximum number of constraints from (6.6), i.e., a solution that minimizes the number of nonzero slack variables. Unfortunately, determining such a solution is known to be NP-complete [9]. ISOMER instead settles for minimizing the sum of the slack variables, because this problem can be solved by linear programming. ISOMER then discards all QFRs that correspond to constraints having nonzero slack. Note that if all the original constraints from (6.6) are satisfiable, then there exists a solution in which all of the slacks equal 0, and hence no QFRs are discarded.

As noted earlier, we want to preferentially discard older QFRs. Instead of minimizing the sum of slack variables, ISOMER therefore minimizes a weighted sum of the slack variables, where the slack corresponding to a QFR is weighted inversely by the age of the QFR. Thus, a QFR that is not satisfied and has nonzero slack incurs a smaller objective-function penalty if it is old than if it is new. Thus an optimal solution is more likely to permit slack in older QFRs, so that such QFRs are preferentially discarded. The age of the $i$th QFR is given by $m - i + 1$. Thus ISOMER solves the following linear program to detect inconsistent
6.5. DEALING WITH DATABASE UPDATES

constraints: Minimize

\[ \sum_{i=1}^{m} \frac{1}{m-i+1}(s_i^+ + s_i^-) \]

subject to (6.9) and (6.10). If \( s_i^+ \) or \( s_i^- \) is nonzero in the resulting solution, then the \( i \)th QFR is discarded. In our implementation, we have used the highly optimized open source Coin LP solver [114]. Discarding QFRs enables ISOMER to merge buckets as described in the next subsection.

Our overall method for eliminating invalidated QFRs is clearly a heuristic. However, as shown in Section 6.8, it works effectively and efficiently in practice. A topic of future work is to enhance ISOMER with a more principled method for eliminating inconsistent QFRs.

6.5.2 Merging Histogram Buckets

Once ISOMER has decided to discard a particular QFR, the total histogram size can potentially be reduced by merging two or more histogram buckets. The process of merging buckets is essentially the inverse of the process for drilling holes described in Section 6.3.3 and is similar to that described in [28].

After the \( i \)th QFR is discarded, ISOMER examines all “top-level” buckets that cover \( R(q_i) \), i.e., all buckets \( b \) such that \( C(b) \subseteq R(q_i) \), but \( C(\text{parent}(b)) \not\subseteq R(q_i) \). Bucket \( b \) can be merged with another bucket in the following cases:

- If bucket \( b \) has exactly the same referring set as its parent, i.e., if \( \{j|C(b) \subseteq R(q_j)\} = \{j|C(\text{parent}(b)) \subseteq R(q_j)\} \), then \( b \) can be merged with its parent. This is because both \( b \) and \( \text{parent}(b) \) have the same number of tuples per unit volume in the maximum-entropy solution; cf. (6.7). The children of \( b \) now become the children of \( \text{parent}(b) \).

- If there is a sibling \( b' \) of \( b \) such that (i) \( b' \) and \( b \) have the same referring set and (ii) \( \text{box}(b') \cup \text{box}(b) \) is hyperrectangular, then \( b \) and \( b' \) can be merged. The children of \( b \) are recursively examined to see if they can be merged with their new siblings, i.e., children of \( b' \). The new merged bucket is also examined to see if it can be merged with any of its siblings.
6.6 Meeting a Space Budget

Since a histogram is meant to be a summary of the underlying data distribution, it typically should not be too large, perhaps a few disk pages. This space budget is placed on the histogram to ensure efficiency when the histogram is read at the time of optimization. At an extreme, if there were no such limit, the entire data itself could be used as a (perfectly accurate) histogram, defeating the purpose of statistics. The addition of new QFRs causes the size of ISOMER’s STHoles histogram to grow as new holes are drilled. Whenever the histogram size exceeds the space budget, ISOMER reduces the histogram size by discarding “unimportant” QFRs. Intuitively, a QFR is unimportant if it provides little information over and above what is already provided by other QFRs. Discarding unimportant QFRs reduces the total histogram size by triggering the merging of buckets (Section 6.5.2).

**Example 6.6.1.** In the example of Section 6.3.2, suppose that we have space to store only two histogram buckets. After adding the QFRs \( N = 100 \) and \( N(make = 'Honda') = 80 \), the resulting histogram has two buckets, and is shown in Figure 6.6(a). Each bucket has volume equal to 2, and bucket \( b_2 \) is a hole in the top-level bucket \( b_1 \). Suppose that we now add the third QFR, \( N(make = 'BMW', color = 'white') = 10 \). ISOMER drills a hole corresponding to this QFR, and the resulting histogram, shown in Figure 6.6(b), has three buckets, violating the space budget. Notice, however, that the addition of this third QFR yields no extra information: because tuples are assumed to be uniformly distributed within a bucket, the histogram in Figure 6.6(b) is already implied by the histogram in Figure 6.6(a). Thus the third QFR is unimportant, and can be discarded.

How can the unimportant QFRs be efficiently determined? Note that the age of a QFR—which ISOMER uses as a criteria for deciding which QFRs are invalid—is not relevant for deciding importance. E.g., in Example 6.6.1, ISOMER can receive many instances of the third QFR in succession, thus making the second QFR very old. However, the second
QFR is still more important than the third QFR.

An elegant aspect of ISOMER is that the maximum-entropy solution yields, for free, an importance measure for each QFR. This leads to a very efficient procedure for detecting and discarding unimportant QFRs. Specifically, ISOMER uses the quantity $|\ln(\lambda_i)|$ as the importance measure for the $i$th QFR, where $\lambda_i$ is the multiplier defined in (6.7). To justify this choice intuitively, we note that if $\lambda_i = 1$, then removal of the $i$th QFR does not affect the bucket counts, so that the final maximum-entropy solution is unaltered. For instance, $\lambda_3 = 1$ in the final solution in Example 6.6.1—see Figure 6.6(b)—and we concluded that the third QFR was unimportant. By the same token, if the multiplier corresponding to a QFR is close to 1, then removal of that QFR will affect the final solution less than if we remove a QFR whose multiplier is much greater than 1. Thus, the $i$th QFR is unimportant if $\lambda_i \approx 1$, i.e., if $|\ln(\lambda_i)| \approx 0$.

An alternative justification for our definition follows from the fact that $|\ln(\lambda_i)| = |y_i|$, where $y_i$ is the Lagrange multiplier corresponding to the $i$th constraint in the maximum-entropy optimization problem (Section 6.4.3). It is well known from optimization theory [26] that the Lagrange multiplier for a constraint measures the degree to which the constraint affects the optimum value of the objective function. Thus $|y_i|$ measures how much the $i$th constraint affects the entropy, i.e., the amount of information in the distribution. In other words, $|y_i| = |\ln(\lambda_i)|$ is a measure of the amount of information carried by the $i$th constraint, and hence a measure of the importance of the $i$th QFR.

Thus, on adding a new QFR, if the histogram exceeds the space budget, ISOMER proceeds by examining the current maximum-entropy solution and, for each $i$, computes the importance measure for the $i$th QFR as $|\ln(\lambda_i)|$. ISOMER then discards the QFR with the least importance according to this measure and merges buckets as described in Section 6.5.2. ISOMER then incrementally computes the new maximum-entropy solution and repeats the above procedure until the histogram is sufficiently small.

### 6.7 Summary of ISOMER

Figure 6.7 gives a complete overview of the ISOMER algorithm. When ISOMER is invoked, it adds all the QFRs present in the query feedback store to the histogram. The QFRs are added in batches whose size is given by the chunk parameter. After the addition of each batch of QFRs, the histogram is adjusted for consistency and size. The chunk parameter
Algorithm *ISOMER*(chunk)

**chunk**: Number of QFRs to be added in a batch  
**φ**: Set of QFRs (persisted in offline store)

**Initialization**:
1. initialize STHoles histogram with bucket \(b_1\), \(C(b_1) = \mathcal{S}\)
2. \(\phi = \{u(b_1) = N\}\)

**On invocation of ISOMER**:
1. read histogram from database stats, \(\phi\) from offline store
2. while (more QFRs remaining in Query Feedback Store)
3. for \(i = 1\) to \(chunk\) do
4. get a new QFR and add it to \(\phi\)
5. drill holes for new QFR (Section 6.3.3)
6. delete inconsistent QFRs from \(\phi\) (Section 6.5.1)
7. merge buckets (Section 6.5.2)
8. compute maximum-entropy solution (Section 6.4.3)
9. while histogram size > space budget
10. delete least important QFR from \(\phi\) (Section 6.6)
11. merge buckets
12. recompute maximum-entropy solution
13. persist final histogram to database, \(\phi\) to offline store

---

affects the running time of ISOMER as follows. Most of the ISOMER cost lies in steps 6–12 of Figure 6.7, i.e., solving the linear program in step 6, and repeated iterative scaling to recompute the maximum-entropy solution in the loop of steps 9–12. If the value of *chunk* is very low, then this cost is incurred after addition of every few QFRs, which tends to increase the overall running time. However, a high value of *chunk* is not desirable either. If the value of *chunk* is very high, then the histogram has a large number of buckets at the beginning of step 6 (due to the addition of a large number of QFRs). Consequently, a much bigger linear program needs to be solved in step 6. Also, the loop in steps 9-12 must be executed a larger number of times to bring the histogram back to within the space budget. Experimentally, we found that a moderate value of *chunk* between 15 and 20 performs best in practice.
6.8 Experiments

In this section, we provide an experimental validation of ISOMER. We prototyped ISOMER on top of the IBM DB2 commercial database, which supports gathering of query feedback. The experiments were conducted on an Intel 2.3GHz processor with 1GB RAM. We used ISOMER to maintain a multidimensional histogram using query feedback and compared ISOMER’s cardinality estimates against estimates obtained by a state-of-the-art commercial optimizer. The only form of multidimensional statistics presently supported by the optimizer is a count of the number of distinct values in a group of columns. We also compared the ISOMER estimates with those obtained by STGrid, since both algorithms use the same type of feedback. We did not compare ISOMER with STHoles, because the feedback collection strategy used by STHoles has a very high overhead that makes the approach impractical, particularly when index scans are used during query processing; see Section 6.10.

Our experiments demonstrate the following:

1. ISOMER provides significantly better cardinality estimates than either the commercial optimizer technique or STGrid.

2. For static data, the cardinality estimates provided by ISOMER consistently improve and then stabilize as QFRs are added to the histogram, i.e., there is no oscillation in accuracy when more QFRs are added (and others removed to keep within the space budget).

3. For changing data, ISOMER learns the new data distribution much faster (i.e., with the addition of many fewer QFRs) than STGrid and provides significantly better cardinality estimates.

4. The overall overhead of ISOMER is low.

For our experiments we used a DMV (Department of Motor Vehicles) database that was derived from a real-world application. For our experiment, we focused on the Cars table which has several correlated columns such as make, model, color, and year. The maximum number of distinct values in any column was approximately 150.

We generated a collection of queries referred to as training queries, issued them to the query execution engine, and collected QFRs for the filters in these queries. We used ISOMER (or STGrid for comparison) to maintain a multidimensional histogram on the set of referenced attributes. The histogram was initialized using one-dimensional database
statistics on the attributes and by assuming independence between the attributes. QFRs collected during execution of the training queries were then used to refine this histogram while allowing a maximum of \( k \) buckets, where \( k \) was an experimental parameter.

To test the accuracy of the maintained histogram, we generated a collection of 200 test queries from the same distribution as that of the training queries. We periodically tested the histogram’s accuracy by comparing the actual and estimated cardinalities for each filter in the test queries. We measured the relative error in the estimation as

\[
\text{Relative Error} = \frac{|N(q) - \hat{N}(q)|}{\max(100, N(q))},
\]

where \( N(q) \) and \( \hat{N}(q) \) are the actual and estimated number of tuples respectively that satisfy the filter \( q \). In our formula, the quantity 100 is a sanity constant [71] that prevents the relative error from blowing up in case of highly selective filters, i.e., filters for which \( N(q) \) is either very small or equal to 0. We measured the overall accuracy of the histogram by the average relative error across all filters in the test queries.

We discuss the accuracy of ISOMER on static data in Section 6.8.1 and then consider updates in Section 6.8.2. We study the running time of ISOMER in Section 6.8.3.

### 6.8.1 Accuracy on Static Data

#### 6.8.1.1 2D Histograms

To demonstrate the ability of ISOMER to handle both numerical and categorical attributes, we used ISOMER to build a two-dimensional histogram on the attributes \texttt{model} (categorical) and \texttt{year} (numerical) of the Car table. Both test and training queries were of the form

\[
\text{SELECT } * \text{ FROM Car WHERE model } = x \text{ AND year BETWEEN } y_1 \text{ AND } y_2
\]

where \( x, y_1, \) and \( y_2 \) are variables. The queries were generated according to the data distribution. First, \( x \) was chosen randomly from all possible models; the probability for choosing a given model was proportional to the model’s frequency in the database. Then \( y_1 \) and \( y_2 \) (\( y_1 \leq y_2 \)) were chosen randomly from between the minimum and maximum years that occur with model \( x \). Note that the execution of such queries gave us QFRs not only for two-dimensional filters, but also for one-dimensional filters, since filters are applied one by one. Thus, we obtained QFRs such as \( N(\texttt{model} = x) \) and \( N(\texttt{model} = x, y_1 \leq \texttt{year} \leq y_2) \).
Figure 6.8: Error of 2D Histogram on model and year

Figure 6.8 plots the error of the various approaches as a function of the number of training queries. Each histogram was allowed to have a maximum of \( k = 175 \) buckets. It can be seen that ISOMER significantly outperforms STGrid by providing much more accurate cardinality estimates. Also note that the error of ISOMER consistently decreases and then stabilizes, but never increases with the addition of more QFRs. Although STGrid also seems to possess this desirable property in case of static data, we show in the sequel that, for dynamic data, the error of STGrid may actually increase with addition of more QFRs. As expected, the optimizer performs poorly, mainly because of the limited multidimensional statistics it supports. That is, the optimizer keeps only a count of the number of distinct values for a group of columns, which is not useful for predicting the cardinality of range filters on the numerical attribute year. Thus, the optimizer estimates are simply based on the independence assumption.

6.8.1.2 3D Histograms

In this experiment, we built a three-dimensional histogram on the highly correlated attributes make, model, and color. Both the training and test queries were of the form

\[
\text{SELECT * FROM Car WHERE make = } x \text{ AND model = } y \text{ AND color = } z
\]
CHAPTER 6. BUILDING HISTOGRAMS WITHOUT DATA ACCESS

Figure 6.9: Error of 3D Histogram on make, model and color

where $x$, $y$, and $z$ are variables. The query-generation process was similar to that described in Section 6.8.1.1. First, a make $x$ was chosen from the various makes according to the data distribution. Then from the various models corresponding to make $x$, a particular model $y$ was chosen, again according to the data distribution. Finally a color $z$ was chosen from the various colors corresponding to the (make, model) combination $(x, y)$. ISOMER obtained QFRs for filters of dimension 1, 2, and 3.

Figure 6.9 plots the error of the various approaches against the number of training queries. Each histogram was allowed to have a maximum of $k = 250$ buckets. ISOMER again outperforms STGrid by an even greater margin than in the 2D case, for two reasons. First, there are no partially overlapping buckets, because all attributes are categorical and hence the only filters are equality filters. Thus no bucket splits take place in ISOMER, so that a larger number of QFRs can be added to the histogram before the space budget is reached, thereby boosting accuracy. Second, STGrid performs poorly because it merely adjusts the tuple counts in the buckets based on QFRs and does not use the QFRs to restructure the buckets. This lack of restructuring is a problem because buckets are determined from the initial one-dimensional statistics, that ignore the correlation between attributes. Since make, model, and color are much more strongly correlated than model and year, the performance of STGrid for this 3D histogram is worse than for the 2D histogram in
6.8. EXPERIMENTS

6.8.1.3 Effect of Space Budget

To study the effect of the space budget on histogram accuracy, we used a 2D histogram on `model` and `year` as in Section 6.8.1.1. Figure 6.10 plots the error of the ISOMER and STGrid histograms against the maximum number of buckets allowed, along with the optimizer error (the optimizer presently maintains a hardcoded, fixed number of buckets). The error was measured after 400 training queries had been executed and the corresponding QFRs used to refine the histogram. As expected, the error of both ISOMER as well as STGrid decreases as more buckets are allowed to be stored. However, ISOMER improves much more rapidly than STGrid as the space budget increases and outperforms STGrid at every value of the space budget.

6.8.2 Accuracy on Changing Data

To study the ability of ISOMER to deal with changing data, we worked with a 2D histogram on `model` and `year` as in Section 6.8.1.1. We interspersed the execution of training queries with the issuing of updates to the database. To specify the type of updates issued, we first define the notion of a *correlated tuple* and a *uniform tuple*. A correlated tuple
is a tuple drawn from the real data distribution; it has all the correlations that the real
database has. For example, since make and model are correlated, a correlated tuple can
have the (make, model) value (Honda, Civic) but never (Toyota, Civic). In contrast,
a uniform tuple is generated by choosing the value of each attribute randomly and inde-
dependently from the attribute’s domain. For example, unlike a correlated tuple, the value
of a uniform tuple can be any (make, model) combination.

We report the results of two experiments with updates. In the first experiment, we
changed the data distribution from uniform to correlated, and in the second, from corre-
lated to uniform.

**Uniform to Correlated:** Specifically, we started out with a database consisting completely
of uniform tuples. The optimizer gathered statistics over this database; these initial statis-
tics remained fixed throughout the experiment. The training queries were executed and
the gathered QFRs used to refine the histogram over time. After every 300 training queries,
20% of the tuples were updated from uniform to correlated tuples. This process was re-
peated 5 times, after which the entire database consisted only of correlated tuples.

Figure 6.11 plots the error of the various approaches against the number of queries exe-
cuted. The spikes in the error curve that occur after roughly every 300 queries correspond
to the times at which the database is updated. All the histograms start off very accurately
because the data is uniform with independent attributes. For all approaches, the error increases when the data is updated. For ISOMER, however, the histogram evolves as queries are issued against the new data distribution, and the error decreases again. The STGrid error also decreases to some extent as the histogram is refined after updates. However, the improvement decreases as data becomes correlated. In fact, when more than 80% of the tuples have been updated, the addition of new QFRs tends to increase the overall error in STGrid. This is mainly due to the heuristic nature of STGrid, that does not preserve consistency when using new QFRs to refine the histogram. As expected, the optimizer error increases at each database update and never decreases, since the optimizer statistics are never refined.

ISOMER is not able to attain its original accuracy after updates have occurred. However, this phenomenon is expected because ISOMER imposes the uniformity assumption on any regions of the data distribution about which it lacks information. Hence ISOMER is bound to be more accurate on uniform than on correlated data.

Correlated to Uniform: We started with a database consisting of only correlated tuples. After roughly every 300 queries, 20% of the tuples were updated from correlated to uniform tuples. Figure 6.12 plots the error of various approaches against the number of queries executed. The results are similar to those in the previous experiment. The error for both ISOMER and STGrid increases when the database is updated. However, on subsequent refinement, the STGrid error decreases only slightly whereas the ISOMER error decreases to a level far below that of STGrid. The optimizer works with statistics that are gathered initially, when the data is correlated. Because the optimizer makes uniformity assumptions in the absence of information, the optimizer error decreases at first as the data starts to become uniform. As the data becomes increasingly uniform, however, the optimizer error starts to increase, because the pernicious effects of outdated statistics (gathered over correlated data) become predominant.

6.8.3 Running Time

The results in Sections 6.8.1 and 6.8.2 show that in contrast to STGrid, ISOMER is robust and highly accurate across various data distributions and update scenarios. Nevertheless, the STGrid approach is very efficient because it uses a simple heuristic to refine the histogram. Although ISOMER is not as efficient as STGrid, we show in this section that the
robustness of ISOMER comes at an affordable cost.

ISOMER’s cost has two components. The first component is the cost of collecting QFRs, which has been shown [113] to be low (less than 5% overhead). Because this cost is incurred by any reactive approach, we do not consider it further. The second component of the ISOMER cost is the cost of refining the histogram using QFRs; this cost is incurred whenever ISOMER is invoked. To study this latter cost component, we work with a 2D histogram on model and year as in Section 6.8.2 and use changing data.

Figure 6.13 plots the total running time of ISOMER to add 4000 QFRs against the maximum number of buckets allowed in the histogram (A comparison with STGrid is not shown, since its running time is essentially 0). The chunk parameter (Figure 6.7) was set to 20. Note that although the worst case complexity is quadratic in the number of buckets (since all sibling pairs may be compared for bucket merging; Section 6.5.2), the actual running time is only slightly superlinear and remains well under a minute even for a moderately sized histogram with 350 buckets. In general, the number of buckets depends in a complicated manner on the number of distinct data values and the smoothness of the data distribution.

We found that most of this time is spent in the process of reducing the histogram size.
6.9. IMPLEMENTING ISOMER FOR A WSMS

Although the basic architecture and techniques used in ISOMER carry over for a WSMS, there are a number of research issues that must be addressed before ISOMER can become a complete solution for the profiling component of a WSMS.

Recall that in Section 6.3.1, ISOMER assumes knowledge of the space $S$ in which the tuple values might lie. To calculate $S$, ISOMER uses the minimum and maximum value of that attribute (for numerical attributes), or the number of distinct values of that attribute (for categorical attributes). For a DBMS, this knowledge about individual attributes is usually present in the system catalogs. However, in the web services scenario, even this information has to be gathered over time by observing query feedback. A comprehensive proposal to do this is an item of future work, but one reasonable possibility is to keep track of the portion of the space $S$ that has been observed so far, and use it as a substitute for the
real space $S$. This approach is justified because we do not have any information about the distribution of tuples outside the observed space $S$, and hence even including that space in the histogram will yield no accuracy benefits (unlike in a DBMS where we at least know the total number of tuples in $S$). The portion of $S$ observed so far can be calculated by recording, for numerical attributes, the minimum and maximum values observed so far, and for categorical attributes, the number of distinct values observed so far (which can be done succinctly using distinct sampling [64]).

Another obstacle to fully implementing ISOMER in a WSMS, and to some extent even in a DBMS, is the presence of overspecific QFRs, i.e., QFRs having filters over other attributes in addition to those on the histogram attributes. For example, suppose we are building a histogram on attributes $\text{make}$ and $\text{model}$ using query feedback. Suppose many queries having filters on these attributes are executed, but all of these queries also have the filter $\text{color} = \text{white}$. Further suppose that the filter on $\text{color}$ is the one to be executed first. Then, even though we shall obtain a large number of QFRs WITH information about the $\text{make}$ and $\text{model}$ distribution, this information is specifically for white cars and cannot be incorporated in the histogram (since there is no bucket specifically for white cars, and there cannot be one unless $\text{color}$ is also a histogram attribute).

The same effect can occur in a WSMS. Recall Example 5.2.3, and suppose we are building a histogram on credit rating and payment history. On running the query in Example 5.2.3, even though we will obtain information about the distribution over credit rating and payment history, this information will specifically be for the list $L$ of individuals that we started with. One possible approach in this scenario might be to store several histograms, each qualified by the list $L$ for which it holds ($L$ may be succinctly stored as a Bloom filter [25]). Then, given a new list $L'$, the selectivities of the filters on credit rating and payment history can be calculated based on the overlap on $L'$ with the lists for the various histograms.

### 6.10 Related Work

Existing work on multidimensional statistics can be broadly classified as addressing either the problem of deciding which statistics to build or that of actually building them. For deciding which statistics to build, workload-driven techniques were proposed in [35], sampling-based approaches in [79], and query feedback-driven techniques in [4]. These
techniques are limited to detecting correlated columns. Since multidimensional statistics are increasingly problematic for higher dimensions, there has also been work [48, 87] on dividing the set of attributes into subsets, so that multidimensional statistics need to be built only for these subsets; the interaction between the subsets being captured through a graphical model. Our work addressed only the latter problem of actually gathering statistics. Many types of statistics have been proposed, e.g., histograms [105] and wavelet-based synopses [95]; we restricted our attention to histograms.

For building multidimensional histograms, proactive approaches that involve a data scan have been proposed, e.g., MHist [105], GenHist [72], and others [48, 99, 119]. As mentioned before, data scans may not effectively focus system resources on the user’s workload and do not scale well to large tables. In principle, histograms can be constructed faster using a page-level sample of the data [33], but large sample sizes—and correspondingly high sampling costs—can be required to achieve sufficient accuracy when data values are clustered on pages and/or highly skewed.

The idea of using query feedback to collect the statistics needed for estimating cardinality was first proposed in [37]. The specific approach relied on fitting a combination of model functions to the data distribution; the choice of functions is ad hoc and can lead to poor estimates when the data distribution is irregular. Query feedback is also used in [113], but only to compute adjustment factors to cardinality estimates for specific filters, and not to build histograms. STGrid [3] and SASH [87] both use query feedback to build histograms, but they often have low accuracy because their heuristic methods for adding new QFRs to the histogram do not maintain consistency. ISOMER’s use of the well founded maximum-entropy principle [108] avoids this problem.

STHoles [28] is another approach that uses query feedback to build histograms. The histogram structure of STHoles is superior to other bucketing schemes such as MHist [105], and for this reason is used by ISOMER. Unfortunately, the original STHoles maintenance algorithm requires, for each query and each histogram bucket, the computation of the number of rows in the intersection of the query and bucket regions. These detailed row counts, which are used to decide when and where to split and merge buckets, are usually not obtainable from the original query filters alone. The query engine must therefore insert artificial filters that specify the (possibly recursive) bucket boundaries. As the number of histograms and the number of buckets per histogram grows, the overhead of evaluating this “artificial feedback” becomes so high as to make the STHoles maintenance approach
impractical. (ISOMER, in contrast, needs only the actual feedback that naturally occurs during query execution—namely, the number of rows processed at each step during the query plan—which can be monitored with low overhead [113].) Finally, STHoles, unlike ISOMER, does not provide principled methods for addressing issues of inconsistent feedback and limits on the available memory for storing the histogram.

The principle of maximum entropy has been used in [94], but for the significantly different problem of consistently estimating the selectivity of conjuncts of filters.

6.11 Conclusions

In this chapter, we addressed the problem of building histograms over multiple attributes without directly accessing the data. Such a method of gathering statistics is the only feasible one in a WSMS where direct data access is typically not available. This method can also be desirable for traditional databases since repeated data access for gathering statistics is expensive for large tables and contends with query execution resources. We developed ISOMER, a new algorithm that maintains multidimensional histograms not by accessing the data, but by obtaining feedback from the query execution engine. ISOMER uses the information-theoretic principle of maximum entropy to refine the histogram based on query feedback gathered over time. Unlike previous proposals for feedback-driven histogram maintenance, which lack either robustness (e.g., STGrid [3]), or efficiency (e.g., STHoles [28]), ISOMER is both reasonably efficient and robust. ISOMER is able to efficiently adapt to changes in the underlying data by quickly detecting and discarding old, inconsistent feedback. In addition, ISOMER uses a principled approach to retain only the most important feedback thereby maintaining the histogram within a prescribed space budget.

We performed a thorough experimental evaluation of ISOMER where we demonstrated that ISOMER can consistently provide more accurate cardinality estimates than existing techniques both for static data, as well as for data with updates. Our experiments also demonstrated that ISOMER incurs low overhead.
Chapter 7

Conclusion

7.1 Thesis Overview

This thesis addresses several challenges pertaining to efficient query processing in two modern data management scenarios: data streams and web services. We introduced these scenarios in Chapter 1. Data Stream Management Systems (DSMSs) are often constrained in terms of resources such as memory and computation. One possible method of reducing resource consumption is to share resources among multiple queries. To this end, in Chapter 2, we addressed sharing of computation among multiple continuous queries, where each query is a conjunction of possibly expensive filters and each filter may occur in multiple queries. We showed that as filter costs increase, the optimal strategy adaptively chooses the next filter to evaluate based on the results of filters evaluated so far, as opposed to evaluating the filters in some fixed order. We presented a greedy adaptive execution strategy and proved that its performance is guaranteed to be close to optimal.

Even with sharing in a DSMS where there are many queries over rapid data streams, resources may not be sufficient to answer all queries exactly at all times. Fortunately, many applications can tolerate an approximate result. Unlike Chapter 2 where we focussed on computation, in Chapter 3, we focussed on memory and addressed the problem of computing approximate answers to continuous sliding-window joins when the available memory may be insufficient to keep the entire state required by the join. The goal of approximation can be to provide either a maximum-size subset of the join result or a random sample, e.g., for aggregation. We proposed models of data and stream arrival that real-life streams often adhere to, and gave optimal algorithms for both types of approximation under these
models.

In Chapter 4, we moved to the setting of a distributed monitoring application, where data is acquired at low-capability devices and then transmitted through a hierarchy of nodes having progressively increasing computational power and network bandwidth. Given a continuous query with expensive filters over the acquired data, we gave an algorithm to find the optimal network node at which each filter should be evaluated so that the overall resource consumption is minimized.

In Chapter 5, we turned to the web services scenario and proposed a general-purpose Web Service Management System (WSMS) that enables querying multiple web services in a transparent and integrated fashion. We addressed the problem of optimizing Select-Project-Join queries spanning multiple web services and gave an algorithm to arrange a query’s web service calls into a pipelined execution plan that optimally exploits parallelism and minimizes the query’s running time. Surprisingly, the optimal plan can be found in polynomial time in contrast to traditional query optimization where the analogous problem is NP-hard.

Since optimization requires statistics on the data, and web services do not usually allow direct access to the data behind them, in Chapter 6 we addressed the problem of gathering statistics without accessing the data. Our solution gathers statistics over time by observing the results of queries over the data. We showed that our approach has a strong theoretical foundation, and unlike previous feedback-based approaches, is both robust as well as efficient.

### 7.2 Future Directions

We suggest directions for future work for both the data streams scenario (Section 7.2.1) as well as the web services scenario (Section 7.2.2).

#### 7.2.1 Data Streams

DSMSs were proposed because traditional DBMSs could not handle the immediate-response processing requirements of applications over rapid continuous streams of data. With continuous proliferation in the amount of data being gathered, and more and more demands for complex analysis, DSMSs are bound to face stringent resource constraints in the presence of high-volume streams. Beyond the steps of sharing, approximation, and
operator-placement described in this thesis for handling resource constraints, the following two areas offer future work in addressing the resource issue:

**Extension to Clusters**

One way to ameliorate resource constraints is to increase the hardware: using a large cluster of machines rather than a single one to run the DSMS. However, to fully exploit a cluster, query operators need to be placed on appropriate nodes to minimize communication and achieve load-balancing between the machines. The possibility of sharing further complicates the operator-placement problem since operators may share state only when they execute on the same machine. Overall, we need a comprehensive solution that can optimize the combined execution of a collection of continuous queries over a cluster by appropriate operator placement.

Apart from increasing the maximum throughput that a DSMS may provide, another reason for using a cluster may be to reduce latency: the time between the arrival of a data element and the time at which it is reflected in the query results. In general, the algorithms in this thesis have focussed on throughput, and new algorithms are needed to address the latency metric. For example, given a query that is a conjunction of filters, one possible way to decrease latency for an incoming tuple is to evaluate multiple different filters on the tuple in parallel on different machines of the cluster, a possibility that we have not considered (since this approach tends to reduce throughput).

**General Approximation**

We have addressed approximation for a very specific class of queries. In order to address approximation for more complex queries, we need to (a) devise approximation schemes for additional operators as we did for joins, and (b) understand how approximation propagates through operators, i.e., how the approximation of an operator is affected when given an approximate result from another operator as input. Given solutions to these problems, we can then devise schemes to allocate resources among different operators of a query plan (or even among different query plans) so that the overall system accuracy is maximized.
7.2.2 Web Services

While the algorithms in this thesis may form the basis of a WSMS query planner (Chapter 5) and profiler (Chapter 6), we believe they only scratch the surface of what promises to be an exciting new research area. Below are some of the most important directions for future work.

Monetary Costs of Web Services

In our work, we have assumed that web service calls incur no cost other than time to execute, and that the objective of query planning is to minimize the overall query running time. However, many real web services may have a monetary price that is charged according to different schemes, e.g., a fixed amount per call, or a fixed amount until a certain threshold. Now, in addition to running time, the query planner must consider the total amount of monetary cost that a particular query plan incurs in answering the query. Furthermore, the two objectives of running time and cost may be at odds with each other, e.g., if there is a fast but expensive web service and a slow cheap one involved in the same query.

Given these two objectives of running time and cost, the planner may need to minimize one subject to a constraint on the other, or more generally optimize a user-specified combination of the two. The algorithms will almost surely depend on which one of several possible pricing schemes is in use, thereby leaving ample scope for future work in this area.

Incorporating Quality of Service (QoS)

Our algorithms currently assume that web services have a constant average response time. In reality, there may be considerable variance in the response times of web services depending on various factors such as the current load on the web service (causing queuing delays) and the current network load. At the same time, web services, especially those with monetary costs, may be able to provide certain quality of service (QoS) guarantees. Thus, it would be beneficial to develop query planning algorithms that can incorporate this QoS information and arrive at plans that will be robust to response-time variations as long as the QoS guarantees are met.
In the absence of QoS information, a solution is to use adaptive query planning mechanisms that can respond to changes in web service response times by rearranging the web services in the query plan. Moreover, the response time of a web service, or the QoS offered by a web service, may be a function of how much money is paid per invocation, thereby connecting this problem to that of monetary costs.

**Multiple Web Services**

In our work, we have assumed there is only one web service that can provide a particular set of data. In reality, there may be several “competing” web services providing the same or overlapping data, perhaps with different QoS, and perhaps at different monetary costs. Here QoS includes not only the response time of the web service, but also other factors such as the quality of the data provided and the availability of the web service. Hence the query execution engine must choose between multiple competing web services to find the best plan according to the chosen optimization metric. The execution engine may even decide to invoke multiple competing web services simultaneously either to boost performance, to obtain a more complete answer (if individual web services often provide incomplete data), or for fault tolerance (if some web services may be intermittently unavailable).

Since multiple web services may have different QoS, and different QoS may be obtained for different monetary costs, the three problems of multiple web services, QoS, and monetary costs need to be addressed together for a fully general solution.

**Query Execution Optimizations**

Our query planning algorithm currently finds a single optimal plan for all tuples. An important next step is to extend the algorithms and the execution engine to allow different tuples to follow different plans (in effect, simultaneously executing multiple plans), a technique that is known to improve performance in the distributed case [41, 84, 121].

Another step that can potentially provide big performance improvements is caching of web service results. Caching saves the overhead of web service calls when the same data is required by a subsequent query. Besides the usual problem of maintaining cache consistency in the face of updates, our query planning algorithms would need to be modified to incorporate caching, e.g., a bottleneck web service may not remain the bottleneck if most of its cost can be avoided due to a good cache hit ratio.
Bibliography


[125] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-join 
queries over streaming information sources. In Proc. of the 2003 Intl. Conf. on Very 


[129] T. Yan and H. Garcia-Molina. The SIFT information dissemination system. ACM 

[130] V. Zadorozhny et al. Efficient evaluation of queries in a mediator for websources. In 


Inc., 1949.