

Exploring Time Related Issues in Data Stream Processing

Wu Ji

B.Eng. (Hons.), Nanyang Technological University

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2010

Acknowledgements

This thesis would not have been completed without the help from a lot of people who support me throughout my PhD journey. I would like to express my deep gratitude to them.

I am extremely grateful to my advisor, Prof. Tan Kian-Lee, for his guidance, patience and encouragement. He is the person who introduced me to the world of database research and guided me through every stage of my PhD study. Despite his busy schedules, Prof. Tan always manages to squeeze time to meet me when I need a discussion or want to seek his advice. His vast experience and knowledge in database systems are invaluable assets to my research. Besides, I also learned from him how to become a better person. To me, he is not just a research mentor, but also a life mentor.

I am also grateful to Dr. Zhou Yongluan. As my senior, he really sets a good role model. His attitude and passion for research greatly influence me. I indeed appreciate his useful advice and constructive feedbacks on my work.

I would like to thank Prof. Chan Chee Yong, Prof. Stephane Bressan and Prof. Panos Kalnis for their continuous help on my research starting from my QE. I thank Prof. Ooi Beng Chin and Prof. Tung Kum Hoe, Anthony for teaching me database and data mining knowledge.

I want to thank Prof. Karl Aberer for hosting me at EPFL for six months so that I had the opportunity to work on an interesting scientific data management project. His insight and vision about database systems inspired me immensely. My thanks also go to Prof. Marc Parlange, Olivier Couach, Hendrik Huwald, Vincent Luyet and Daniel Nadeau for their discussions about scientific data

research.

I thank all my friends and the students in the database lab: Bao Zhifeng, Cao Yu, Chen Su, Chen Yueguo, Dai Bingtian, Hui Mei, Lin Yuting, Liu Xuan, Wang Nan, Wu Huayu, Wu Sai, Wu Wei, Xiang Shili, Yang Xiaoyan, Yu Tian, Zhang Dongxiang, Zhang Jingbo, Zhang Zhenjie and many others. They helped me a lot in my daily life. And their presence makes my PhD journey a fun and memorable experience.

Last but not least, I would like to thank my wife, Wantong, for her unfailing love and constant support. And I am deeply indebted to my parents and parents-in-law. I would not have come this far without their unconditional love and care.

The past few years have witnessed a surge of data in the form of streams such as network traffics, stock updates and monitoring information from sensor devices. The fast, time-varying and unbounded nature of data streams, however, challenges the traditional database management paradigm which is intended for store-based data only. The new Data Stream Management System (DSMS) has been proposed by the database community to tackle new issues arising from processing persistent queries running over these continuous data. One can say that a DSMS query is a DBMS query extended in time domain. This implies that both input and output of a DSMS query are better to be modeled as functions of time rather than static values or sets. This observation leads us to study DSMS with the emphasis on time, the critical aspect that distinguishes traditional query processing from stream query processing.

In the first piece of work, we study time issues on stream input. As data is only accessible in sequential manner in stream processing, the input sequence hence becomes crucial. Most stream data are naturally sorted according to the time when they are generated. Such a temporal order, however, is often scrambled for various reasons as the data are transmitted over the network. A scrambled tuple order poses a significant challenge on memory management for stateful operations (such as join) as these operations require a huge amount of memory space to buffer the received input in order to absorb the impact due to tuple disorder. Traditionally, memory management for these operations is query-driven: a query has to explicitly define a window for each (potentially unbounded) input to bound the size of the buffer allocated for that stream.

However, output produced this way may not be desirable (if the window size is not part of the intended query semantic) due to the volatile input characteristics. We propose a new data-driven memory management scheme which explores the intrinsic properties of stream input to intelligently allocate buffer space. Results show that our new scheme not only improves the query result accuracy but also significantly reduces the memory overhead.

Time also plays an important role in stream output. Data stream applications often involve time-critical tasks such as disaster early warning, network intrusion detection and online financial analysis. These applications impose very strict requirements on the timeliness of output delivery. Experience shows that the traditional operator-based stream scheduling strategies may not always be sufficient to fulfill such real-time requirements. In the second piece of work, we focus on tuple-based stream scheduling that features fine-grained resource control to meet these timing requirements. By drawing an analogy between tuple scheduling and job scheduling, we propose several effective resource allocation strategies inspired by the classic job scheduling problem. We also compare the pros and cons of each strategy and discuss their applicability under different scenarios.

The last piece of work is devoted to a case study of data stream applications. We built a scientific sensor data processing engine with the aim to integrate data streams collected from heterogeneous sensor stations and offer a unified data platform to query, analyze and visualize sensor information to facilitate scientific research and data exploration. Time issues discussed in the previous works are revisited in the context scientific data stream processing to appreciate their significance in better understanding stream processing characteristics and,

consequently, how they can be leveraged to improve system performance in practice.

To summarize, we use time as the key to approaching several important issues in DSMS. Both the experiments and the case study show that our proposed algorithms and strategies are effective in boosting the performance of data stream processing.

1	Introduction	1
1.1	Time in Data Stream Systems	3
1.2	Time Related Issues in Stream Processing	5
1.2.1	Memory overhead	5
1.2.2	Output timeliness	6
1.3	Contributions	7
1.3.1	Data-driven Memory Management for Stream Join . . .	7
1.3.2	Tuple-based Data Stream Scheduling	8
1.3.3	Scientific Sensor Data Management: A Case Study . . .	10
1.4	Thesis Outline	10
2	Literature Review	13
2.1	Stream Query Processing Overview	13
2.2	Important Data Stream Operations	15
2.2.1	Sliding Window Operation	15
2.2.2	Stream Join	17
2.3	Adaptive Query Processing	19
2.4	Sequence Database	20
3	Data-driven Memory Management for Stream Join	21
3.1	Introduction	22
3.2	Preliminaries	24
3.2.1	Problem Statement	24
3.2.2	Intra-stream Delay	26

3.2.3	Inter-stream Delay	28
3.3	Memory Cost Model	32
3.3.1	Joining Synchronized Streams	34
3.3.2	Joining Unsynchronized Streams	35
3.4	Issues at Query Level	37
3.4.1	Pipelined Join on the Same Attribute	39
3.4.2	Pipelined Join on Different Attributes	41
3.5	Memory-Constrained WO-Join	45
3.5.1	Memory-Sort First Strategy	46
3.5.2	Disk-Buffer First Strategy	47
3.5.3	Hybrid Approach	48
3.6	Experimental Study	49
3.6.1	Experimental Evaluation	51
3.7	Related Work	59
3.8	Summary	60
4	Tuple-based Data Stream Scheduling	63
4.1	Introduction	64
4.2	Motivation and Challenges	68
4.2.1	Motivating Example	68
4.2.2	Challenges	69
4.3	Preliminaries	70
4.3.1	Metric Definition	70
4.3.2	System Model	72
4.3.3	Problem Statement	73

4.3.4	Related Work on Data Stream Scheduling	73
4.4	From Stream Scheduling to Job Scheduling	75
4.4.1	Job Cost, Due Date and Utility	76
4.4.2	Related Work on Job Scheduling	78
4.5	Applicability of Data Stream Scheduling	79
4.6	Greedy Strategies	80
4.6.1	Basic Strategy	81
4.6.2	Improving Scheduling Accuracy	83
4.7	Deadline-Aware Strategies	96
4.7.1	Deadline-Dominant Strategy	97
4.7.2	Profit-Dominant Strategy	100
4.8	Intelligent Tuple Batching	101
4.9	Experimental Evaluation	102
4.9.1	Experimental Setup	102
4.9.2	Performance Study	104
4.10	Strategies in Retrospect	111
4.11	Summary	113
5	Scientific Sensor Data Management: A Case Study	115
5.1	Background	116
5.2	Related Work	118
5.3	Motivating examples	120
5.3.1	Scenario One	120
5.3.2	Scenario Two	122
5.4	Data Model	123

5.5	Operations	126
5.5.1	Perspective Construction	127
5.5.2	Relationship Between Perspectives	129
5.5.3	Operators	130
5.5.4	Illustrative Example	136
5.6	Query Execution Strategies	137
5.7	Optimization Techniques	138
5.7.1	Preprocessing and Query Rewrite	139
5.7.2	Optimizing Query Execution	140
5.7.3	Optimization for Visualization	146
5.8	Experimental Setup	150
5.8.1	Implementation	150
5.8.2	Dataset	152
5.8.3	Query Set	152
5.9	Performance Evaluation	153
5.9.1	Routine Query Execution	153
5.9.2	HyperGrid vs. Array-based Implementation	154
5.9.3	Query Rewrite	155
5.9.4	Buffering Strategy	156
5.9.5	Optimizing Execution Strategy	158
5.9.6	Visualization Optimization	159
5.10	Data-driven Memory Management for HyperGrid	161
5.10.1	Implementing Dynamic Base Perspectives	161
5.10.2	Using Data-driven Memory Management	163
5.11	Multi-Query Scheduling in HyperGrid	164

5.12 Summary	169
6 Conclusions	171
6.1 Summary of Contributions	172
6.2 Future Work	173

List of Figures

1.1	DBMS processing paradigm Vs. DSMS processing paradigm . . .	2
3.1	Example of synchronized streams	33
3.2	Buffer requirements for S_i to join with synchronized streams (the worst case scenario)	34
3.3	Buffer requirements for S_i to join with unsynchronized stream (the worst case scenarios)	36
3.4	Example of pipelined join on the same attribute	38
3.5	Example of pipelined join on different attributes	38
3.6	Example queries used in this chapter	50
3.7	Buffer space for streams in Query 1.	52
3.8	Buffer for S_7 (Query 2)	54
3.9	Buffer for S_8 (Query 2)	54
3.10	Buffer for S_7 (Query 3)	55
3.11	Buffer for S_8 (Query 3)	55
3.12	WO-Join vs W-Join	56
3.13	Memory reduction strategies	56
3.14	Average memory consumption	57
3.15	Output latencies	58
4.1	Example query in foreign exchange trading	68
4.2	A query graph example	72
4.3	From stream scheduling to job scheduling (Input I_1)	75
4.4	From stream scheduling to job scheduling (Input I_2)	75

4.5	Illustration of the <i>OptProfit</i> algorithm	94
4.6	Coefficient u (DD)	104
4.7	Coefficient u (PD)	104
4.8	Tuple batching	106
4.9	Operator sharing	107
4.10	Query weight variance	108
4.11	Response to input load	109
4.12	Response to tuple urgency	109
4.13	Response to bias	110
5.1	Work flow of Example 5.3.1	120
5.2	Illustration of the query execution in Example 5.3.1	136
5.3	HyperGrid (HG) Vs. Array (AR)	153
5.4	Effect of query rewrite	155
5.5	Effect of buffer strategy	156
5.6	Optimizing execution strategy (total runtime)	157
5.7	Optimizing execution strategy (average response time)	158
5.8	Plot with 15% result computed (using directed random walk al- gorithm)	159
5.9	Plot with 100% result computed	159
5.10	Run time cost for visualization	160
5.11	Illustration of a dynamic base perspective	162
5.12	An example of multi-query graph	166
5.13	Comparison of update frequency	168
5.14	Comparison of weighted update frequency	168

List of Tables

3.1	Important notations used in this chapter	26
4.1	Comparison between OBS and TBS	64
4.2	Important notations used in the algorithm	85
5.1	Default settings for perspective parameters	130
5.2	Characteristics of perspectives for different operators	130
5.3	Query set description	152
5.4	Average memory saving with the data-driven memory manage- ment scheme	164

1

Introduction

The past few years have witnessed a surge of data in the form of streams. Compared to traditional finite set-based data, streaming data offers a more natural way to model continuous processes in the physical world (such as temperature variation) as well as long running human activities (such as currency exchange trading) in daily life. The fast, time-varying and unbounded nature of data streams, however, challenges the traditional database management paradigm which is intended for store-based data only. For this, a new database management system, called Data Stream Management System (DSMS), has been proposed and developed by the database community in recent years with the aim to more efficiently handle queries running over continuous streams. Compared to traditional Database Management System (DBMS), DSMS mainly differs in the following ways:

1. Queries in DSMS are typically running continuously as new data is flowing in while queries in DBMS are snapshot queries. This also implies query

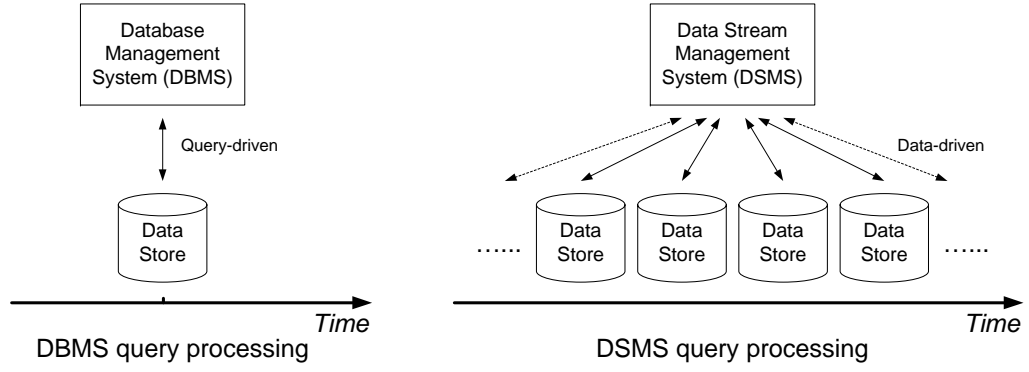


Figure 1.1: DBMS processing paradigm Vs. DSMS processing paradigm

execution in DSMS is data-driven, as opposed to being query-driven in DBMS.

2. In most DSMSs, data is only accessible in a sequential manner while in DBMS both sequential and random access are possible.
3. A query evaluation scheme for stream processing must be dynamic and adaptive to the ever changing input characteristics, which are unpredictable in nature. In contrast, a query evaluation plan in DBMS only deals with processing data with static attributes.

In short, the difference between a query in DBMS and that in DSMS is as follows: A DSMS query can be viewed as a DBMS query extended in time domain. Such an extension implies that both input and output of a stream query become functions of *time* rather than static values or sets. Figure 1.1 gives a graphical illustration of such a difference.

In view of this, our approach to the design of DSMS concentrates on various issues surrounding *time*, the critical aspect that distinguishes DSMS query

processing from DBMS query processing. As we shall see later, many new challenges that emerge in DSMS relate to the notion of time in one way or another.

1.1 Time in Data Stream Systems

The notion of time can be found in almost all important components of data stream processing. These include:

- Input

Different from DBMS which only manages data sets, a DSMS mainly manages data sequences (in addition to data sets). The key distinction between data set and data sequence is that the latter can be ordered. And for the majority of the data sequences seen in stream applications, the ordering key is time. Typically, streaming data is either timestamped or attached with some type of temporal ordering (e.g. sequence number or epoch). Such information is crucial as the results of many stream queries depend on it. For example, in an environmental monitoring application, a temperature reading generated by a sensor at time t could entail something very different from the same reading reported with another timestamp value. Note that temporal ordering can be defined in various ways depending on the user specifications and application scenarios. Two popular approaches are: 1) ordered by when they are generated by the data source 2) ordered by when they enter the DSMS. In most cases, sequences produced according to these two approaches are not identical, especially for distributed applications where data transmission delay is substantial.

- Query

One significant difference between a conventional DBMS query and a DSMS query is that a DSMS query usually includes a window clause for each stream input involved. For example:

```
SELECT AVG(T.temp_val) FROM TEMPERATURE[Range 60 minute] AS T
```

The above query computes the latest hourly average temperature from the “TEMPERATURE” stream. As the input stream is potentially unbounded, a window clause is essential to define a finite subset of the input where the current query result is computed. The most common type of window is the sliding window, which shifts along the time line. A window definition consists of two components: a reference point in time and a window size. By default, the reference point is “now”, which means the window ends at the current time. The window length could be measured in terms of the maximum predetermined number of tuples (called count-based window) or a fixed time period (called time-based window). The example above belongs to the latter. The clause in the square bracket defines a time window of 60 minutes. It means the average value is computed only using tuples received in the recent 60 minutes.

- Output

The continuous query (CQ) processing paradigm of DSMS particularly suits real-time data applications where computed output streams out as new input continuously flows in. Examples of such applications include on-line stock analysis and network intrusion detection, etc. Owing to

the real-time nature, these applications usually have a very stringent requirement on the timeliness of output delivery. Consider on-line stock analysis as an example. Because the query results depend on the current stock price, they have to be produced almost instantaneously to diminish the impact due to stock price fluctuation. In many stream systems, output latency is considered the most important type of Quality-of-Service (QoS).

1.2 Time Related Issues in Stream Processing

Similar to traditional data management, strategies or techniques proposed for DSMS mostly focus on either both or one of the following objectives: 1) to reduce various costs or overheads associated with data processing 2) to improve key performance metrics (such as throughput or output latency). However, compared to DBMS, to achieve the above goals in DSMS becomes much more difficult owing to the highly dynamic and unpredictable nature of streaming data and the demanding requirements specific to stream applications. This thesis covers two important aspects pertaining to data stream processing, one for each of the above-mentioned objectives. Unsurprisingly, the notion of *time* plays a central role in both topics.

1.2.1 Memory overhead

Efficient memory management has always been an important concern in data management. But in DSMS, the issue becomes more pronounced due to its unique data access pattern. By default, stream data is only accessible in a

sequential manner. This means if an operation involves random access or anything more than a single scan of the input, then all the data have to be buffered in the main memory before they become irrelevant to the query results. Given that the size of stream input can be very huge or even unbounded, such a processing pattern poses a significant challenge on efficient memory usage. Existing window-based query offers a straightforward way to constrain memory overhead. However, this is not always a good option. We will discuss in Chapter 3 how to exploit the time information attached to the input tuples in order to better utilize limited memory space.

1.2.2 Output timeliness

As mentioned, for stream applications that perform time-critical tasks, the output quality does not only depend on the answer accuracy. More importantly, it depends on how timely the output is generated. Hence, output latency becomes one of the main Quality of Service (QoS) metrics in DSMS. Different from DBMS where the query response time simply corresponds to the query evaluation time, the output delay perceived by users in DSMS is also influenced by the time of input availability. Developing a query execution scheme that ensures the timeliness of output delivery can be very challenging since it involves various factors: query complexity, query priority, coordination among inputs (for multi-input query only), and system utility, etc. In Chapter 4, we analyze the issue and present scheduling strategies that aim to optimize the responsiveness of output in a multi-query data stream environment.

1.3 Contributions

The main contribution of this thesis lies in the in-depth analysis of time-related issues in stream processing. The objectives are to minimize data processing overhead and to improve key performance metrics for stream applications through a better understanding of how *time* plays a role in DSMS. The study of *time* in stream input inspires us to develop a new stream join strategy that minimizes memory overhead. The study of *time* in stream output leads us to discover several novel stream scheduling algorithms for improved QoS performance. We also implemented a scientific sensor data processing system as a case study for these issues in a real life scenario.

1.3.1 Data-driven Memory Management for Stream Join

As mentioned, memory overhead has always been a critical issue in stream processing. This is particularly true for queries involving stateful operators such as join. Traditionally, the memory requirement for a stream join is *query-driven*: a query has to explicitly define a window for each (potentially unbounded) input to bound the size of the buffer allocated for that stream. However, output produced this way may not be desirable (if the window size is not part of the intended query semantic) due to the volatile input characteristics. Moreover, the *query-driven* approach often leads to extremely inefficient memory utilization. Our proposed solution well addresses this issue. Specifically,

- We introduce the concept of *data-driven* memory management and contend that, whenever possible, memory allocation for stream join is better

off being data-driven than being *query-driven*.

- Following the concept of *data-driven* memory management, we propose a new stream join processing paradigm, called *Window-Oblivious Join* (WO-Join), which is able to dynamically adjust the memory buffer size based on the current input data characteristics.
- Extensive experimental study suggests that WO-Join significantly outperforms traditional windowed join in terms of both output quality and memory-efficiency.

The details about data-driven memory management is presented in Chapter 3. A primary version of this work was published in [100]. And later an extended version appeared in [101].

1.3.2 Tuple-based Data Stream Scheduling

In this piece of work, we study the problem of on-time delivery of stream output, a topic which has been largely overlooked before. It was believed that the traditional operator-based scheduling techniques are sufficient to address issues arising from the real-time requirements of output generation in DSMS. Unfortunately, this is not always the case. For time-critical applications whose success depends on the prompt delivery of each output result, a tuple-level resource control is mandatory. That explains why good operator-based resource allocation strategies that significantly improve system related performance metrics (e.g. average processing cost or total memory overhead) may not do well in terms of user-oriented metrics, such as timeliness of output delivery as well as other

QoS measures. Compared to operator-based scheduling, tuple-based scheduling is a less studied but more challenging topic. The main difficulty comes from the fact that the number of stream tuples are enormous. Hence, fine-grained tuple-level resource control is almost impossible due to the prohibitive overhead associated. Our approach towards tuple-based scheduling is unique: By drawing an analogy between tuples and jobs (as in real-time job scheduling), we translate a tuple scheduling problem to a job scheduling problem. Such a new vision allows us to find some very good scheduling strategies that could not have been discovered otherwise. Contributions of this work include:

- Identification of Tuple-Based Scheduling (TBS) as an important class of stream scheduling,
- An in-depth analysis of how TBS problem can be transformed into a job scheduling problem,
- Presentation of two general approaches to data stream scheduling, namely greedy strategy and deadline-aware strategy. Within each approach, two algorithms are proposed with the aim to improve the overall performance from a job scheduling perspective,
- Extensive experimental studies that identify factors that could influence the effectiveness of scheduling strategies and compare the performance of our proposed scheduling solutions.

Part of this work was published in [102] while the remainder was reported in [99]. Chapter 4 merges these two portions and provides a complete description about our tuple-based stream scheduling strategies.

1.3.3 Scientific Sensor Data Management: A Case Study

The surge of interest in data stream processing in recent years is largely driven by the fast-growing Wireless Sensor Network (WSN) applications that have a profound impact on our life. We have seen different kinds of sensor networks being deployed for a wide range of purposes: environmental monitoring, traffic control, military surveillance, manufacturing quality control, to name a few. It is forecasted that the number of wireless sensor network nodes will reach approximately 120 million units in 2010, with the overall shipment value arriving at about US \$15.0 billion [1]. The last technical contribution of this thesis features a scientific sensor data management system as a case study for data stream processing. The system is built with the aim to integrate data streams collected from heterogeneous sensor stations and offer a unified data platform to query, analyze and visualize sensor information to facilitate scientific research and data exploration. Time issues discussed in Chapter 3 and 4 will also be recapitulated in the context scientific data stream processing to appreciate their significance in better understanding stream processing characteristics and, consequently, how they can be leveraged to improve system performance in practice. This work is presented in Chapter 5, which is a revised version of [103].

1.4 Thesis Outline

The rest of the thesis is organized as follows:

1. Chapter 2 surveys related work, which covers various aspects of data

stream processing including general-purpose stream prototype systems, the state-of-the-art query processing techniques for important stream operations (window-based operation, stream join, etc.) and adaptive query processing, etc. Work on sequence database will also be reviewed.

2. Chapter 3 proposes a novel memory management strategy based on the notion of time associated with each stream input.
3. Chapter 4 discusses time issues related to output production. Several scheduling strategies that aim to improve the output timeliness are proposed.
4. Chapter 5 presents a scientific sensor data management system as a case study to discuss how streaming data is queried and processed in a real situation. It first describes the general framework of the system, and then revisits the time issues addressed in the previous chapters in the context of scientific sensor data processing.
5. Chapter 6 concludes the thesis with a summary of contributions and provides directions for future work.

2

Literature Review

This chapter surveys existing research work that is relevant to this thesis. Work that is related to a specific topic of this thesis will be discussed separately in the respective chapters.

2.1 Stream Query Processing Overview

Stream query processing (or continuous query processing) has been widely studied over the past few years by many research groups. Interest in this area has generated plenty of academic and industrial projects. Some of them are general-purposed systems while others are designed specifically for certain applications.

The *STREAM* system [6] is a general-purpose Data Stream Management System that aims to handle multiple continuous, high-volume, and time-varying streams in addition to managing traditional stored relations. A concrete declarative query language called Continuous Query Language (CQL) [8] was developed to support complicated query semantics such as sliding window ag-

2.1. STREAM QUERY PROCESSING OVERVIEW

gregation or relation-to-stream operation. The focus of this project includes query approximation [9, 68] and dynamic query execution [13, 14]. The *TelegraphCQ* [10, 26, 67] project shares some common data management issues with *STREAM*. However, it emphasizes adaptive query engine for efficient processing in volatile and unpredictable environments. *Aurora* [3, 17] is another well known project, which is targeted exclusively towards stream monitoring applications. *Aurora* adopts a workflow-style specification of queries. As one of its features, all resource management decisions such as scheduling [25] and load shedding [88, 89] within *Aurora* are based on the well-defined QoS specifications.

On the industrial side, the Gigascope project [32] offers a solution for monitoring high speed network streaming data. Similar to *STREAM*, Gigascope has a well-defined stream query language with SQL-like syntax. One distinctive feature of Gigascope is that it breaks a query into smaller pieces so as to push query operations down as far as possible. Simple operations such as filter can even be performed at hardware level. Such strategy greatly reduces the system workload, hence leading to enhanced capability. More recently, Franklin et al. [38] developed a new system called *Truviso* that aims to seamlessly integrate continuous query processing into a full-function database system to meet the needs of new emerging data stream applications.

Other stream related projects that are peculiar to certain application domains include *NiagaraCQ* [27] for efficient processing of streaming XML data, *StatStream* [106] for monitoring financial statistics over many streams and *Tribeca* [87] for managing Internet traffic, etc.

2.2 Important Data Stream Operations

2.2.1 Sliding Window Operation

The introduction of windowed operation makes it possible for blocking operators such as sort and aggregation to be evaluated over unbounded stream data. Essentially, a windowed operator breaks a stream into possibly overlapping subsets of data and computes results over each. The fact that the notion of windowed operation itself provides opportunities for query optimization has been widely recognized in many literatures. A number of techniques are proposed to improve query efficiency by exploiting the window definition and construction.

In [62], the authors classified various types of windows based on the window semantics and proposed a *Window-ID* (WID) approach for query evaluation. The idea is to identify each window extent by a *Window-ID* and create many-to-many relationships between window extents and input tuples involved. So whenever a new tuple arrives, the affected window extents can be easily identified and the corresponding output will be generated automatically. The advantage of this approach is that for some operations (such as aggregation) input tuples just need to be scanned once. They are not required to be buffered (since tuples are processed on the fly as they arrive), which leads to less memory consumption. Another interesting technique is to divide overlapping windows into several disjoint sub-windows [106] or "panes" [61]. Queries are evaluated over these small windows first and then merged together to produce the final output. The advantage of this approach is to avoid duplicate calculations when window extents overlap among each other. Similar idea is adopted on parallel side, two

2.2. IMPORTANT DATA STREAM OPERATIONS

partitioning strategies are proposed in [51] for scalable execution of expensive stream queries: *window split* (WS) and *window distribute* (WD). The *window split* approach is essentially the same as the sub-window idea. The only difference is that now these sub-windows are sent to different nodes for processing. The *window distribute* turns out to be even simpler, where input partitioning occurs just at the logical window level. However, both approaches incur significant overheads. They are only viable for processing expensive scientific queries. The authors in [44] took a unique view of sliding window by studying not only the window semantics defined over the input streams but also the query update patterns as a result of such windowed operations. They studied all commonly used query operators and classified them according to when and how the result tuples are expired as a window slides forward. Based on this observation, they proposed the notion of update-pattern-aware modeling for efficient query processing. Building index on sliding window is also considered in recent work. [42] proposes two types of indices optimized specifically for main-memory sliding windows: one for answering set-valued queries which offers a list of attribute values and their counts; the other for answering attribute-valued queries which provides direct access to tuples. Overhead is a concern here since indices have to be updated while new data flow in.

Improving query efficiency through approximation is another topic of interest. [34] shows how to maintain simple statistics over sliding window and formalizes the space requirement as a function of the length of sliding window and accuracy parameter. In [12], two algorithms are presented namely “chain-sample” and “priority-sample” for sampling input tuples over constant-size windows and variable-size windows respectively. And [45] makes use of

histograms to support incremental maintenance of statistics over a sliding window. Besides, load shedding [33, 39] is also a commonly adopted approach when input data becomes overwhelming. However, if sliding window itself is regarded as an approximation for the entire streaming data, then all the above approaches become "approximating the approximation". Hence, it is sometimes difficult to quantify the accuracy of the results produced by these methods.

2.2.2 Stream Join

Streaming algorithms for join evaluation is another relevant research area. The first of such algorithms is Symmetric Hash Join [98], which was originally designed to allow high degree of pipelining in parallel database systems. XJoin [92] extends Symmetric Hash Join to use less memory by allowing parts of the hash table to be moved to secondary storage. A similar idea also appears in Ripple Join operator [46, 48]. A variation of Symmetric Hash Join was proposed in [93] with the emphasis on processing priority tuples. Viglas et al. [95] developed a multi-way version of XJoin called MJoin. XJoin consists of a tree of two-way joins, which maintains a join subresult for each intermediate two-way join in the plan. While in an MJoin, each relation R has a separate query plan, or *pipeline*, describing how updates to R are processed. New tuples in R are joined with the other $n - 1$ relations in some order, generating new tuples in the n -way join result. Therefore, an MJoin need not maintain any intermediate join subresults. However, experiments in [95] also show that MJoin does not scale well with the increase of the number of join inputs. This suggests that MJoin also needs a query plan tree just like XJoin for optimal performance.

2.2. IMPORTANT DATA STREAM OPERATIONS

The adaptive join ordering problem for stream data was studied in [13]. The *Adaptive Greedy* (or *A-Greedy*) algorithm proposed in this paper dynamically changes the join sequence among the input streams during run-time so that operators with higher selectivity will always be performed earlier. This has been shown to be an effective approach to reducing join processing cost.

Stream join over sliding window has also been extensively studied. Hammad et al. [47] identified various window join scenarios over multiple streams in terms of where the window semantics are defined. For example, a window join over multiple streams could be a chain of pairs of two-way join, or a single join predicate involving multiple streams, etc. This paper introduced a class of join algorithms, each for a different join scenario. Particularly, the paper highlights that unsynchronized input data streams (due to network delay or variation in data arrival rate) could potentially cause inaccurate answers as arrived tuples may get expired before they can completely join with delayed tuples. This issue has a similar flavor to *Referential-Integrity Constraints* as in [15] and has been honored in our proposed optimization model as well. [43] studied several algorithms for sliding window multi-join processing including multi-way incremental nested loop joins (NLJs) and multi-way incremental hash joins. Join ordering heuristics were also proposed. The aim is to minimize the processing cost. Rate-based query optimization is addressed in [94] and [55]. [94] suggested a rate-based estimation approach to optimize the query plan for stream data as opposed to the cardinality-based approach for stored data. Two heuristics, namely *Local Rate Maximization* and *Local Time Minimization*, were proposed to choose the plan with the highest output rate. [55] studied joining streams with different arrival rates. It derived the cost model of performing

window join over streams using both Hash Join (HJ) and Nested Loop Join (NLJ). The experiments indicated that when the probed data rate is low, HJ outperforms NLJ in terms of CPU-efficiency; and the other way around when the probed data rate is high. Therefore, a hybrid of HJ and NLJ approach may be the best choice. Tran et al. [90] proposed an optimization technique for windowed join in conjunction with aggregation. By transforming the query plan so that aggregation is performed before join, a considerable performance improvement can be achieved.

2.3 Adaptive Query Processing

Although dynamic query plan re-optimization has been well studied for static databases (e.g. [31, 50, 54]), these approaches are not capable of handling streaming data either because the statistics required for optimization are only available in set-oriented data or because the query plan cannot evolve and adapt to changes in stream characteristics for a long run.

The work in [24] suggests utilizing the pause-drain-resume paradigm for dynamic plan migration. However, this strategy does not explicitly explain how to handle the case where queries contain stateful operators such as window join with intermediate results. Zhu et al. [105] addressed this issue and proposed an online plan migration strategy for continuous queries with stateful operators. The strategy minimizes the plan migration costs by reusing the states that have been computed in the obsolete plan.

The novel Eddies architecture [10, 35, 67] enables very fine-grained adaptivity by eliminating query plans entirely, instead routing each tuple adaptively

across the operators that need to process it. Eddy's always adapting solution makes it suitable for a highly dynamic environment. However, such flexibility comes with the price of significant per-tuple based processing overheads.

2.4 Sequence Database

We lastly look at related work on sequence database. In some sense, sequence database system can be seen as the predecessor of DSMS. Seshadri et al. [79, 80, 81] formally defined an algebra and a declarative query language for querying ordered relations. They also addressed several important issues concerning the design of a sequence database system. These provide important theoretical foundation for DSMS. However, their work mainly targets one-time, non-continuous sequence data processing. In contrast, a DSMS is expected to process sequence data that are continuous, unbounded and time-varying. Important issues such as query processing efficiency and memory management hence have to be reconsidered.

3

Data-driven Memory Management for Stream Join

Stream query processing is usually memory intensive owing to the fact that most data streams have huge volume and unpredictable data characteristics. Hence, efficient memory utilization has always been an important topic in data stream research. In this work, we focus on a new memory management scheme which leverages timestamp information to considerably reduce the memory overheads. The chapter is organized as follows. We give an introduction in Section 3.1. In Section 3.2, we formulate the problem and identify factors that impact memory consumptions for stream join. Sections 3.3 and 3.4 present the memory cost model at operator level and query level respectively. We extend our techniques to processing queries under memory-constrained scenarios in Section 3.5. Section 3.6 reports the experiment results. Related work is discussed in Section 3.7. Finally, Section 3.8 concludes the chapter.

3.1 Introduction

The emerging data stream applications (such as network intrusion detection, traffic monitoring, and online analysis of financial tickers) often involve processing sheer volume of online data in a time responsive manner. Computations as such are highly memory-intensive, especially for operations that need to maintain run-time states (join, aggregation, etc.). Hence, queries with these operations typically need one or more window clauses, which effectively dictate the amount of run-time buffer required during the query execution. We call this *query-driven* memory management scheme. While such mechanism works well in many situations, there are scenarios where output quality can be severely impaired as the desired answers may be missing from the result set. For example, an input tuple may have already been purged from the memory before it completely joins with tuples from other streams due to the inflexible state buffer size fixed by the query window¹. More results may be obtained if the state buffer size can adapt according to the input characteristics. To make this more concrete, consider a location tracking application (based on localization techniques such as the one presented in [16]) in a wireless sensor network environment. The location of an object (a transmitter) can be inferred by synthesizing the Signal Strength (SS) measured at the surrounding sensors. In such applications, each sensor produces a series of data tuples with uniform schema $(epoch, x, y, z, val)$, (where *epoch* refers to the time when the signal is recorded, x , y and z correspond to the physical coordinates of the sensor, and

¹We recognize that there are applications whereby the specified window is an important part of the query semantics, i.e. the user does not intend to obtain the entire set of the join results, but only certain fraction of them. For example, the user may be only interested in the results generated from the tuples received in the recent 5 minutes. In this chapter, we do not focus on this type of query.

val is the SS measured). A typical query for tracking an object looks like the following:

```
SELECT * FROM    Sensor1 S1, Sensor2 S2,  
                  Sensor3 S3, Sensor4 S4  
WHERE   S1.epoch = S2.epoch  
AND     S2.epoch = S3.epoch  
AND     S3.epoch = S4.epoch
```

In this query, data packets from four sensors are routed to the central location to be joined together before the target location can be predicted. Owing to the unreliable communication channel, which results in a highly dynamic network topology as well as the availability of multiple paths from the source to the centralized location, tuples may experience different transmission delays and therefore arrive at the central location in an arbitrary order (i.e., tuples are not ordered according to their epoch values). Now, as the traditional Window-Join (W-Join) [55] only joins tuples that are within a pre-defined window boundary, it implicitly assumes that all latency and out-of-order effects are absorbed by the window specified by the user. However, this may not hold since users typically have no clue about the underlying input characteristics or the network topology. As a result, query accuracy may drop significantly when packets encounter severe transmission delay or a high degree of order scrambling. The only way to obtain consistent quality results is to define “sufficiently large” query windows, which inevitably leads to extravagant memory overheads that many systems cannot afford. The dilemma of choosing the appropriate window size shows that the W-Join approach is too rigid and therefore not suitable for such applications.

To address the issue, we contend that, whenever possible, memory allocation for stream join should be data-driven instead of query-driven. We therefore propose a new memory management scheme, called *Window-Oblivious Join* (WO-Join), which dynamically determines the state buffer size according to the current data input. WO-Join characterizes a query’s memory requirements as a function of two types of delays: namely *intra-stream delay* and *inter-stream delay*. When these two delays are bounded, complete join results are computable using finite memory space. WO-Join guarantees complete join results when these two parameters are known apriori. If such information is not available beforehand, WO-Join can monitor the two parameters during runtime and allocate the buffer size accordingly to ensure high quality results, even under memory-constrained scenario. Our experimental study demonstrates that WO-Join significantly outperforms W-Join in terms of both output quality and memory-efficiency in many situations.

3.2 Preliminaries

3.2.1 Problem Statement

We consider WO-Join over a set of infinite streams S with equality join predicate. The WO-Join may include one or more MJoin [95] operators. Within an operator, one buffer is maintained for each input stream. The buffer serves as a sliding window for the stream so that input data are inserted and removed in a FIFO manner. The size of each buffer is adjusted dynamically to ensure the join is performed in a memory-efficient way. In this chapter, we first consider

the basic memory management issues presented at operator level. Then we extend our discussion to query level.

To summarize, the problem studied in this chapter is as follows: Given a multi-way equijoin query without explicit window semantic, produce high quality results in a memory-efficient manner. The quality of the results is measured by the output accuracy defined as follows:

$$\text{output accuracy} = \frac{\# \text{ of join tuples actually produced}}{\# \text{ of join tuples that can be produced}} \quad (3.1)$$

Note that when the streams are completely unordered, there is no way to produce high quality join results since the memory evaluation costs for that are unbounded. Here, we focus on applications where join results are bounded-memory computable. Back to our previous sensor network example, although data packets may experience time-varying delays, such delays are bounded as we know sooner or later packets will be delivered to the destination. Since the delays are bounded, tuple ordering on the join key (“epoch”) should be *in the long run* monotonically increasing. This opens the possibility to produce complete (or near-complete) results with limited memory space. In fact, the core issue we address is to relate the memory evaluation costs with the stream characteristics. As can be seen later, this issue is not as straightforward as it may look. We start the discussion by introducing two main sources of memory overheads for stream join, namely the *intra-stream delay* and the *inter-stream delay*.

Notation	Meaning
S	The whole set of input streams
S_i	i th input stream
s_i	A tuple in stream S_i
$s_i.A$	Join key on which stream ordering is defined
k_i	Scrambling Factor of stream S_i
r_i	Data rate of stream S_i
$PI(s_i)$	Physical Index of tuple s_i
$VI(s_i)$	Virtual Index of tuple s_i
$VI_{min}(s_i)/VI_{max}(s_i)$	Minimum/Maximum value from the set $VI(s_i)$
$L_{S_i \hookrightarrow S_j}(t)$	Lag from S_i to S_j at time t
$M_{S_i \hookrightarrow S_j}(t)$	Memory space to buffer S_i tuples w.r.t. stream S_j at time t
$Mul(S_i)$	Multiplicity of stream S_i
$DSF_{A \rightarrow B}$	Dependent Scrambling Factor from column A to column B

Table 3.1: Important notations used in this chapter

3.2.2 Intra-stream Delay

Intra-stream delay causes tuples' order to be scrambled within a stream (the tuple ordering issue). To facilitate our study, we first define what is a totally ordered stream, then quantify a partially ordered stream by a parameter called *Scrambling Factor* (SF). For ease of exposition, assume each tuple in the stream has an index, called *Physical Index* (or PI), which corresponds to the arrival position of that tuple in the stream. For example, the PI of the first arrived tuple from a stream is 1. The PI of the next arrived tuple is 2 and so on. A tuple s_i 's PI value is given by the function $PI(s_i)$. Important notations used throughout this chapter are listed in Table 3.1.

A totally ordered stream is thus defined as follows:

Definition 3.2.1 *A totally ordered stream S_i must fulfill the following condi-*

tion for any pair of tuples s_i and s'_i from S_i :

$$\text{If } PI(s_i) < PI(s'_i), \text{ then } s_i.A < s'_i.A$$

A partially ordered stream with *scrambling factor* k is defined as follows:

Definition 3.2.2 A partially ordered stream S_i , with **Scrambling Factor** k , must fulfill the following condition for any pair of tuples s_i and s'_i from S_i :

$$\text{If } s_i.A \leq s'_i.A, \text{ then } PI(s_i) - k \leq PI(s'_i)$$

where k is the minimum integer that satisfies the inequality.

Notably, our notion of totally ordered stream defines a strictly ordered sequence, i.e. no duplicates are allowed. If tuples with the same attribute value exist in the stream, such stream is only considered partially ordered even though it is in *non-descending* order². The reason is that, as we shall see later, from memory management point of view tuples with duplicate values do affect memory requirements as if they are unordered. Therefore, we do not distinguish between tuples that are out of order and tuples with duplicate values in our definition.

Clearly, according to Definition 3.2.2, a lower k implies a stricter ordered sequence while a higher k implies a more scrambled sequence. For the rest of the chapter, we use the value of SF (or k) to measure the degree of out-of-order for a given partially ordered stream.

²Without loss of generality, we only consider ascending or non-descending order. A stream with descending or non-ascending order can be handled similarly.

3.2.3 Inter-stream Delay

Inter-stream delay occurs when streams are not synchronized. Roughly speaking, the inter-stream delay is defined as the arrival time difference between the matching tuples from different inputs. Intuitively, such delay directly impacts the memory consumption: Longer inter-stream delay implies larger memory overheads. However, to judge whether streams are synchronized or to quantify the delay between unsynchronized streams is not trivial, especially when input streams are not totally ordered. In this section, we first discuss how to quantify the inter-stream delay between totally ordered streams, then extend the concept to partially ordered streams.

To ease the presentation, we make the following definition, which will be used throughout the chapter.

Definition 3.2.3 *Given a tuple $s_i \in S_i$, a tuple $s_j \in S_j$ whose join key is equal to $\max\{s'_j.A \mid s'_j.A \leq s_i.A, s'_j \in S_j\}$ is called s_i 's **next-of-kin tuple** from S_j .*

3.2.3.1. Totally Ordered Streams

To begin with, let us consider delays between totally ordered streams. From the memory requirement perspective, we use *Lag* to quantify such delay. *Lag* measures the number of tuples from one stream that the system has to buffer as they may potentially join with tuples that have not arrived from the other stream.

Definition 3.2.4 *Let s_i and s_j be the latest arrived tuples from totally ordered streams S_i and S_j at time τ , then the lag from S_i to S_j , denoted by $L_{S_i \leftrightarrow S_j}(\tau)$,*

is quantified as:

$$L_{S_i \hookrightarrow S_j}(\tau) = PI(s_i) - PI(s'_i) \quad (3.2)$$

where s'_i is s_j 's next-of-kin tuple from S_i .

The intuition is that the arrival of s_j can evict tuples in the S_i buffer whose join keys are less than s_j 's next-of-kin tuple. So only $PI(s_i) - PI(s'_i)$ number of tuples need to be retained in the S_i buffer (remember there are no tuples with duplicate values here since streams are totally ordered). Note that $L_{S_i \hookrightarrow S_j}(\tau)$ can be negative. This occurs when s'_i comes after s_i in S_i .

Another way of understanding this equation is we can treat s'_i as s_j 's “correspondence tuple” in stream S_i . Therefore the *Lag* between the streams can be measured by the distance between s_i and s'_i .

S_i is said to be *synchronized* with S_j at time τ if

$$0 \leq L_{S_i \hookrightarrow S_j}(\tau) \leq 1 \quad (3.3)$$

Furthermore, if $L_{S_i \hookrightarrow S_j}(\tau) > 1$, we say that, at time τ , S_i runs ahead of S_j or S_i leads S_j . S_i is called the “leading” stream and S_j is called the “lagging” stream.

It is evident that, if two totally ordered streams are always synchronized between each other, the buffer space required for both streams are nominal. The arrival of a new tuple from one stream can immediately evict the last arrived tuple from the other stream.

3.2.3.2. Partially Ordered Streams

Comparatively, synchronization between partially ordered streams is much

harder to define. This is because when the next-of-kin tuples arrive at different time, it is difficult to tell whether the time difference is due to delay between the streams or due to scrambled tuple order within the stream. To better understand the synchronization issue in this case, we first need to introduce the notion of *Virtual Index* (VI):

Definition 3.2.5 *The virtual index of a tuple $s_i \in S_i$, denoted by $VI(s_i)$, is the set of $PI(s_i)$ when stream S_i is sorted in non-descending order.*

It is important to note that, different from $PI(s_i)$ which returns a unique index value, $VI(s_i)$ returns a set of consecutive indices. This is because attributes with duplicate values take up multiple positions in the sequence. It is valid to map a tuple's virtual index to any one of these positions. We denote $|VI(s_i)|$ to be the size or cardinality of the set $VI(s_i)$. And let $VI_{min}(s_i)$ and $VI_{max}(s_i)$ be the minimum and maximum value from the set $VI(s_i)$. For a stream S_i whose join key is duplicate-free, $VI_{min}(s_i) = VI_{max}(s_i)$ and $|VI(s_i)| = 1$. An important relationship between a tuple's VI and its PI is summarized below:

Theorem 3.2.1 *For a partially ordered stream S_i with $SF = k$, we have the followings:*

$$PI(s_i) - VI_{min}(s_i) \leq k \tag{3.4a}$$

and

$$VI_{max}(s_i) - PI(s_i) \leq k \tag{3.4b}$$

Proof of Theorem 3.2.1 The theorem can be easily proved by contradiction. For Inequality 3.4a, assume $PI(s_i) - VI_{min}(s_i) > k$ for some tuple s_i . According to the definition of *virtual index*, there are exactly $(VI_{min}(s_i) - 1)$ numbers of

tuples whose join key values are less than that of s_i . Since $PI(s_i) > VI_{min}(s_i) + k$, there are at least $(VI_{min}(s_i) + k)$ numbers of tuples whose PI s are less than $PI(s_i)$. That means there are at least $(VI_{min}(s_i) + k) - (VI_{min}(s_i) - 1) = k + 1$ numbers of tuples whose values are greater than or equal to the value of s_i and whose PI s are less than $PI(s_i)$. Let t denotes one of these tuples, we have $PI(t) < PI(s_i)$. Also according to Definition 3.2.2, $PI(s_i) - k \leq PI(t)$. Therefore, the value of $PI(t)$ can only be an integer in the interval $[PI(s_i) - k, PI(s_i))$. And there are exactly k numbers of such integers within the interval. However, there are at least $(k + 1)$ numbers of such t . That means at least one of these tuples has PI less than $PI(s_i) - k$. This contradicts with the premise that stream S is partially ordered with $SF = k$. Inequality 3.4b can be proved analogously. \square

Now we are ready to derive the function to compute $L_{S_i \hookrightarrow S_j}(\tau)$. Here, the *Lag* measures the number of buffered tuples from S_i caused solely due to the delay between the streams, while the memory overheads owing to intra-stream delay are excluded.

Definition 3.2.6 *Let s_i and s_j be the latest arrived tuples from S_i and S_j respectively at time τ , and s'_j be a tuple from S_j such that $PI(s_j) \in VI(s'_j)$ and s'_i be s'_j 's next-of-kin tuple from S_i . Then,*

$$L_{S_i \hookrightarrow S_j}(\tau) = PI(s_i) - VI_{max}(s'_i) \quad (3.5)$$

The definition of synchronization for partially ordered streams is the same as that of totally ordered streams mentioned earlier (Equation 3.3).

The intuition here is we first find out the tuple s'_j that would have appeared at s_j 's position if the stream is ordered, and then we “map” s'_j to its “correspondence tuple” in S_i as done in the discussion for totally ordered streams. As an example, Figure 3.1 shows a snapshot of two data streams arriving at the system. The timeline on the left indicates the tuple arrival time. The values inside $\langle \rangle$, $[]$ and $\{ \}$ correspond to the join key, PI and VI of the tuple, respectively. For example, at $t = t_5$, tuple $\langle 66 \rangle \in S_1$ arrives, with $PI(\langle 66 \rangle) = p + 2$ and $VI(\langle 66 \rangle) = p + 4$, where p is the PI of the first arrived S_1 tuple shown in the figure. Now consider at $t = t_5$, the last tuple arrived from S_1 and S_2 are $s_1 = \langle 66 \rangle$ and $s_2 = \langle 65 \rangle$, respectively. And the tuple $s'_2 = \langle 63 \rangle$ is the one that satisfies the condition $PI(s_2) \in VI(s'_2)$. Then we find out s'_2 's next-of-kin tuple from S_1 to be $s'_1 = \langle 63 \rangle$ (the tuple arrived either at t_1 or at t_4). Since $PI(s_1) = p + 2$ and $VI_{max}(s'_1) = p + 2$, we get $L_{S_1 \hookrightarrow S_2}(t_5) = 0$ (Definition 3.2.6). Hence the two streams are synchronized at time $t = t_5$. Analogously, we can derive that $L_{S_1 \hookrightarrow S_2}(t_{10}) = 3$, with $s_1 = \langle 67 \rangle$, $s_2 = \langle 68 \rangle$, $s'_2 = \langle 65 \rangle$ and $s'_1 = \langle 64 \rangle$.

3.3 Memory Cost Model

An accurate memory cost model is crucial for WO-Join. It is not only important during query processing to ensure proper memory allocation, but also useful on other aspects such as performing the disk-buffer scheduling to handle severely unsynchronized streams, dynamically distributing memory among different queries, or implementing admission control to avoid memory congestion. As an example, we will demonstrate in Section 3.5 how to use the cost model for disk-buffer scheduling. In this section, We first show how to derive such a

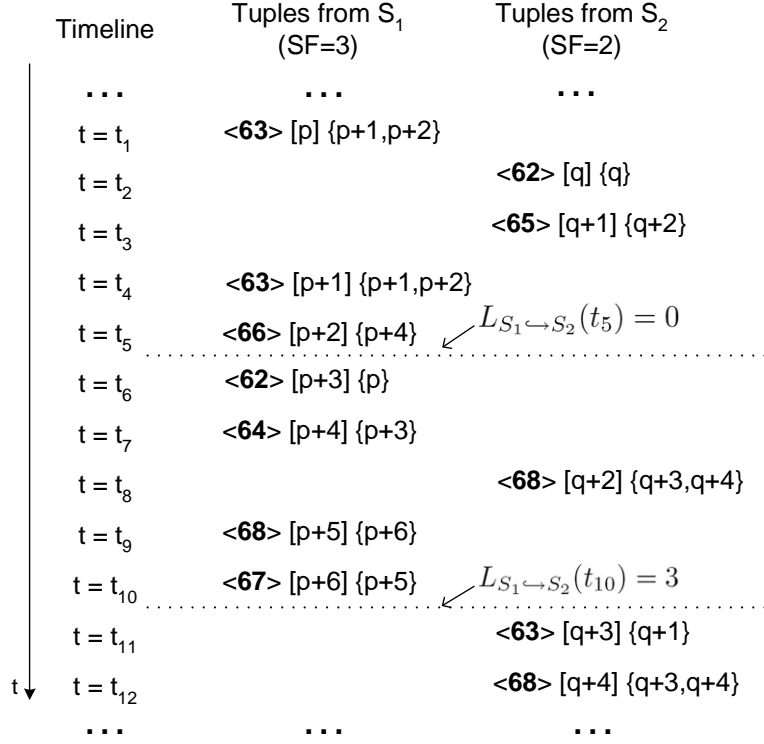


Figure 3.1: Example of synchronized streams

memory cost model for any given stream join operation.

The emergence of MJoin [95] allows multiple inputs to be joined in one step. Compared to traditional binary join, MJoin leverages the memory efficiency by eliminating the intermediate join results. Therefore, we choose to base our memory cost model on MJoin. Binary join can be viewed as a special case of MJoin with two inputs.

As mentioned before, the memory cost model characterizes the memory requirements based on two factors: scrambled tuple ordering (SF) and delay among streams (Lag). Given these two parameters, the model should be able to evaluate the memory costs for generating the complete join results in the worst case scenario. In what follows, we consider the impact of the tuple ordering

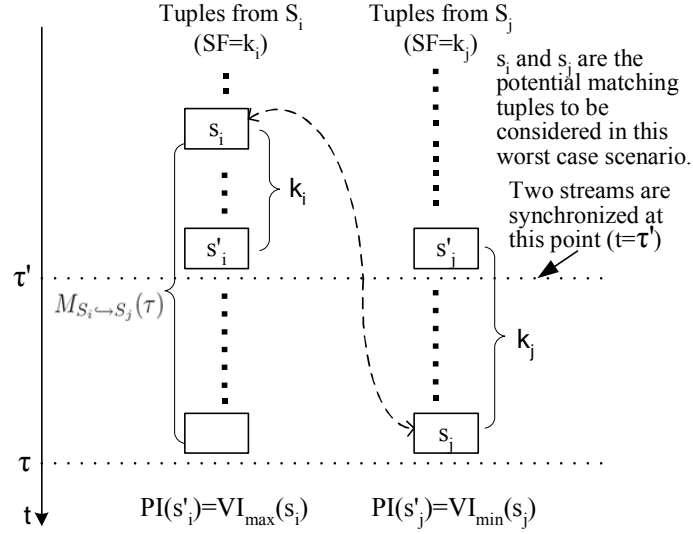


Figure 3.2: Buffer requirements for S_i to join with synchronized streams (the worst case scenario)

alone by first assuming streams are all synchronized (i.e. $Lag = 0$ or 1). Then we extend it to the unsynchronized stream scenario.

3.3.1 Joining Synchronized Streams

Given two streams S_i and S_j , let s_j be the latest tuple from S_j at time τ , and s'_j be a tuple such that $PI(s'_j) \in VI(s_j)$. Suppose the two streams are synchronized at time τ' when tuple s'_j arrived at the system, then the number of tuples from S_i that should be retained in the buffer at time τ , denoted by $M_{S_i \leftrightarrow S_j}(\tau)$, in order to guarantee the complete join results with S_j is given as follows:

$$M_{S_i \leftrightarrow S_j}(\tau) = k_i + r_i \cdot \frac{k_j}{r_j} \quad (3.6)$$

where r_i and r_j are the data rates of stream S_i and S_j , respectively. We illustrate the intuition by the example given in Figure 3.2. The two streams S_i and S_j

are synchronized at time τ' when s'_j arrives. Let s'_i denote the latest tuple from S_i at time τ' and s_i be s_j 's next-of-kin tuple from S_i . We can guarantee that tuples arrived k_i tuples earlier than s'_i have join key values less than $s_i.A$. Therefore, we only need to buffer the last k_i tuples from S_i that arrived earlier than τ' . Furthermore, we also have to buffer tuples from S_i that arrive during the interval $[\tau', \tau]$. The upper bound of this buffer is $r_i \cdot \frac{k_j}{r_j}$ as the number of tuples that arrive from S_j during this period is at most k_j (Theorem 3.2.1).

3.3.2 Joining Unsynchronized Streams

Now consider the memory overheads to join unsynchronized streams. For two streams S_i and S_j , let the latest arrived tuple from S_j at time τ be s_j , and s'_j be a tuple such that $PI(s'_j) \in VI(s_j)$. Suppose the *Lag* from S_i to S_j at time τ' when tuple s'_j arrives at the system is $L_{S_i \leftrightarrow S_j}(\tau')$. Then the number of tuples from S_i that should be kept in the buffer at time τ , denoted by $M_{S_i \leftrightarrow S_j}(\tau)$, is:

$$M_{S_i \leftrightarrow S_j}(\tau) = \max\{1, k_i + r_i \cdot \frac{k_j}{r_j} + L_{S_i \leftrightarrow S_j}(\tau')\} \quad (3.7)$$

The intuition of the above equation is illustrated in Figure 3.3(a) and Figure 3.3(b). Figure 3.3(a) shows the scenario where S_i leads S_j by $L_{S_i \leftrightarrow S_j}(\tau')$ at time τ' . Imagine that if we “remove” the last $L_{S_i \leftrightarrow S_j}(\tau')$ tuples from S_i arrived before τ' , the situation is identical to the synchronized stream scenario illustrated in section 3.3.1: The amount of S_i tuples that should be retained in the buffer is $k_i + r_i \cdot \frac{k_j}{r_j}$. However, since now we need to additionally buffer $L_{S_i \leftrightarrow S_j}(\tau')$ tuples due to the stream lag, the actual buffer size becomes $k_i + r_i \cdot \frac{k_j}{r_j} + L_{S_i \leftrightarrow S_j}(\tau')$.

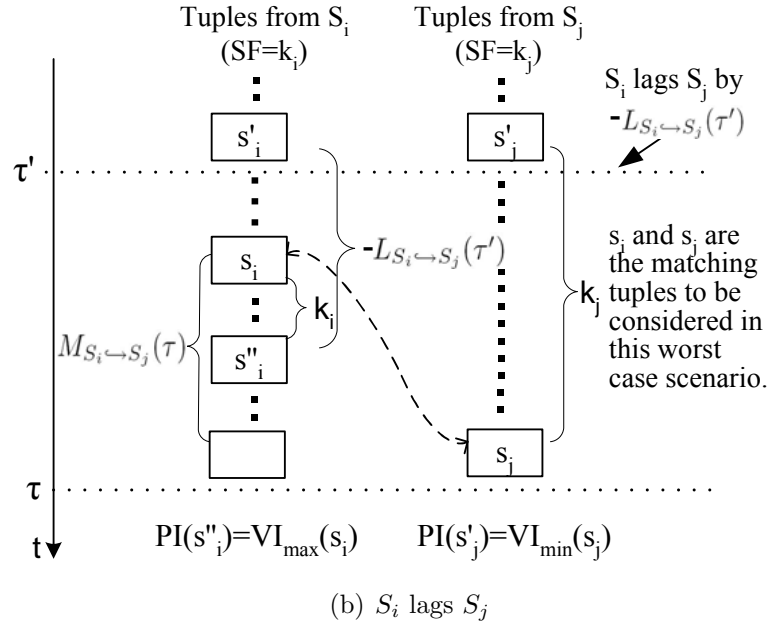
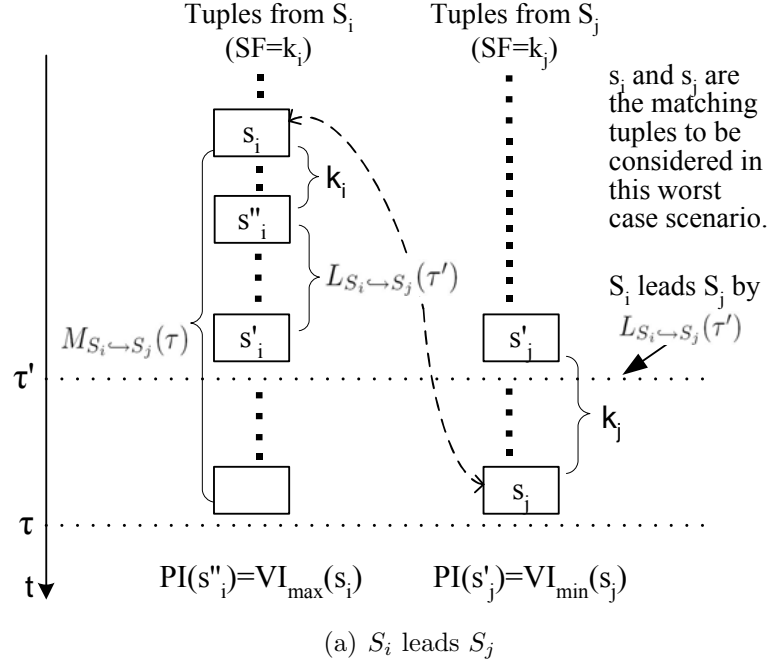


Figure 3.3: Buffer requirements for S_i to join with unsynchronized stream (the worst case scenarios)

Next, let us consider the scenario where S_i lags S_j shown in Figure 3.3(b). Same as before, the number of tuples from S_i that should be buffered for S_j is $k_i + r_i \cdot \frac{k_j}{r_j} + L_{S_i \hookrightarrow S_j}(\tau')$. Interestingly, since now $L_{S_i \hookrightarrow S_j}(\tau') < 0$, this implies the lag between S_i and S_j helps reduce the memory overheads incurred at S_i . Actually, $|L_{S_i \hookrightarrow S_j}(\tau')|$ can be so large that the entire equation turns negative. Physically, it means stream S_i runs far behind S_j such that tuples from S_i can immediately join with all matching tuples from S_j (if such matching tuples exist) when they arrive. In this case, the size of the buffer needed at S_i (for matching tuples from S_j) is minimal. So we set it to 1.

The discussions so far focus on binary join. However, it is straightforward to generalize it to multi-way join scenario. For example, consider the memory requirements to buffer S_i in a three way join $S_i \bowtie S_j \bowtie S_k$. We can calculate the memory requirements between S_i and each of the other input streams (i.e., $M_{S_i \hookrightarrow S_j}(\tau)$ and $M_{S_i \hookrightarrow S_k}(\tau)$). The maximum value will be the actual buffer size required for S_i . Formally, the memory space needed for S_i in an MJoin operator with n input streams is:

$$M_{S_i}(\tau) = \max_{\forall j \in n, j \neq i} M_{S_i \hookrightarrow S_j}(\tau) \quad (3.8)$$

3.4 Issues at Query Level

Section 3.3 analyzes the memory evaluation costs for a single join operator. For a query plan that consists of multiple join operators, the total costs can be evaluated by repeatedly applying the above memory cost model for each operator appeared in the query plan tree. This means besides the input streams,

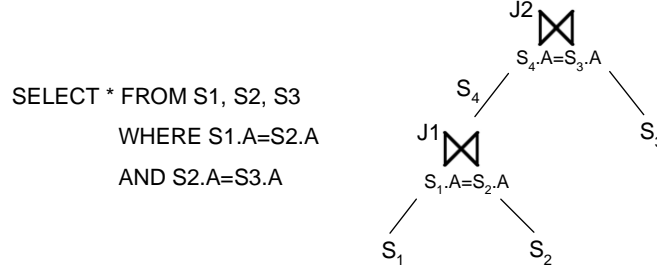


Figure 3.4: Example of pipelined join on the same attribute



Figure 3.5: Example of pipelined join on different attributes

the system also needs to know the data characteristics of the intermediate result streams since they are essential for the cost estimation of the next join operator downstream in the query plan. Take the query in Figure 3.4 as an example. To estimate the memory consumption of operator J2, one has to know the data characteristics of S_4 , such as SF and data rates. We show that these parameters can be derived from the inputs of its upstream operator (i.e. J1's in this example) in the plan.

Before we move on to derive the intermediate stream characteristics, it is important to first distinguish between two types of join queries since they have significant implications on how stream characteristics should be estimated. Figures 3.4 and 3.5 give the examples of these two types of queries. Notice in

Figure 3.4, the downstream operator J2 shares the same join attribute as its upstream operator J1. In Figure 3.5, on the other hand, the downstream operator J2 uses a different join attribute³ from its upstream operator J1. We explain how streams characteristics are estimated under the two different scenarios in the following sections.

3.4.1 Pipelined Join on the Same Attribute

When the upstream and downstream operators share the same join attribute, we would like to know how the stream ordering (w.r.t. that join attribute) and data rate evolve after input streams are joined together. The following theorem addresses this problem by giving the upper bound of the output stream characteristics based on the characteristics of its contributing input streams.

Theorem 3.4.1 *For an MJoin operator with n input streams, the data rate and SF of its output stream, denoted by r_o and k_o , respectively, are bounded as follows:*

$$r_o \leq \sum_{i=1}^n r_i \prod_{\substack{j=1 \\ j \neq i}}^n Mul(S_j) \quad (3.9)$$

$$k_o \leq \sum_{i=1}^n k_i \prod_{\substack{j=1 \\ j \neq i}}^n Mul(S_j) \quad (3.10)$$

where k_i is the SF of stream S_i and $Mul(S_j)$ is the multiplicity of stream S_j .

Here, a stream's *Multiplicity* refers to the maximum size of the tuples with duplicate values. In another words, $Mul(S_j) = \max\{|VI(s_j)| \mid \text{for all } s_j \in S_j\}$.

³The data type of the second join attribute can be timestamp or any other (partially) ordered sequence.

If a stream's join attribute value is duplicate-free, the *Multiplicity* is 1. It is important to note the stream's *SF* is always greater than or equal to its *Multiplicity*, since tuples with duplicate values affect the stream's *SF* according to Definition 3.2.2.

Proof of Theorem 3.4.1 We first prove Inequality 3.9. The output data rate is maximal when tuples from each input stream join maximal number of tuples from all other streams. Given the streams' multiplicities, we can know that a tuple from S_i can join with no more than $\prod_{\substack{j=1 \\ j \neq i}}^n \text{Mul}(S_j)$ number of tuples from other input streams. Given S_i 's data rate r_i , the maximum number of outputs produced due to arrival tuples from S_i is $r_i \prod_{\substack{j=1 \\ j \neq i}}^n \text{Mul}(S_j)$ per unit time. In the worst case scenario, for a short period of time, it is possible that outputs produced due to arrival tuples from different streams do not overlap. In this case, the total output rate is therefore the summation of output rate generated due to tuple arrivals from each individual input stream. Hence,

$$r_o \leq \sum_{i=1}^n r_i \prod_{\substack{j=1 \\ j \neq i}}^n \text{Mul}(S_j).$$

We continue to prove Inequality 3.10. Let us consider binary join first, say $S_i \bowtie S_j$. Given two pairs of matching tuples s_i/s_j and s'_i/s'_j , where s_i matches s_j , s'_i matches s'_j and $s_i \leq s'_i$. To consider the scenario where the output *SF* is maximum, let the pair s'_i/s'_j join as early as possible and the other pair join as late as possible. Due to the ordered constraints on S_i and S_j , s'_i can arrive at most k_i places before s_i , and s'_j can be at most s_j places before s_j . For each tuple that is between s'_i and s_i , it could maximally join with $\text{Mul}(S_j)$ number of tuples from S_j . Similarly, for each tuple that is between s'_j and s_j , it could maximally join with $\text{Mul}(S_i)$ number of tuples from S_i . If all these join results are produced after s'_i joins with s'_j and before s_i joins with s_j . Then,

the maximum value of k_o will be the distance between the s'_i/s'_j pair and s_i/s_j pair in the output stream, which is $k_i \times Mul(S_j) + k_j \times Mul(S_i)$. Generalize the above, we get for an MJoin operator with n input streams, the SF of the output stream, k_o is bounded by $\sum_{i=1}^n k_i \prod_{\substack{j=1 \\ j \neq i}}^n Mul(S_j)$. \square

Compared to SF or data rate, inter-stream delay between the intermediate result stream and other input streams is much easier to quantify since it is simply determined by the “bottle-neck” stream, i.e. the most “lagging” input among all contributing streams. For example, if in Figure 3.4, S_1 is the lagging stream between S_1 and S_2 , then it is sufficient to determine the Lag between S_4 and S_3 by just measuring the Lag between S_1 and S_3 using the techniques introduced in Section 3.2.3. (This is based on the premise that the processing delay of operator J1 is negligible compared to the amount of Lag between input streams, which is the case most of the time; otherwise processing delay incurred on J1 should be taken into account when measuring the Lag)

3.4.2 Pipelined Join on Different Attributes

For this type of query, data rate of intermediate stream can be estimated the same way as introduced in Section 3.4.1 because the value is not influenced by the change of the join attribute on the next operation. For inter-stream delay, the approach mentioned in the previous section can be applied as well except that now Lag is measured with reference to the new attribute which participates in the next join operation. Hence, the main issue here is to determine the new intra-stream delay (SF) of the intermediate stream. Since the next join is preformed on an attribute different than the one used in the previous operation,

the tuple ordering has to be correspondingly redefined on the new join key. For the example in Figure 3.5, the ordering for stream S_2 is first defined on attribute $S_2.A$. However, after J1 is performed, the ordering w.r.t. attribute $S_2.A$ is no longer relevant. Instead, we are now interested in the ordering scrambling factor w.r.t. attribute $S_2.B$. As we shall see, to find out the SF on one attribute after join is performed on another attribute is not trivial. We show how this can be achieved below.

3.4.2.1. Dependent Scrambling Factor

The problem described above can be rephrased as follows: Given a stream which has two partially ordered columns, if the tuple sequence has been altered such that the SF defined on one attribute is changed to a new value, say k' , how would the SF defined on the other attribute be affected? Intuitively, to answer this question we have to establish certain correlation about the orderings of these two columns. We therefore introduce the notion of *Dependent Scrambling Factor* (or DSF) here. The definition is given below:

Definition 3.4.1 *Given a stream with two partially ordered columns A and B , the **Dependent Scrambling Factor** from column A to column B , denoted by $DSF_{A \rightarrow B}$, is the SF on column B when the stream is sorted on column A .*

Basically, $DSF_{A \rightarrow B}$ measures the degree of order-scrambling on column B with ordered column A as the reference sequence. Here, we use DSF to characterize the ordering correlation between two columns. Although such characterization may not completely capture the relationship between two (partially) ordered sequence, it suffices for our purpose. Generally speaking, a smaller $DSF_{A \rightarrow B}$ implies a stronger tuple order dependency from column A to B while

a larger $DSF_{A \rightarrow B}$ implies a weaker order dependency between the two columns. It is important to note that DSF is a property that describes some inherent relationship between two columns of the same stream. Hence, its value is not influenced by the change of the actual tuple sequence.

3.4.2.2. SF Derivation

With the notion of DSF , we can proceed to derive the SF of one column when the SF defined on the other column is known. The theorem below gives the details:

Theorem 3.4.2 *For a stream with two partially ordered columns A and B , if the SF on column A is k_A and the Dependent Scrambling Factor from A to B is $DSF_{A \rightarrow B}$, then the SF on column B , denoted by k_B , is bounded as follows:*

$$k_B \leq 2 \cdot k_A + DSF_{A \rightarrow B} \quad (3.11)$$

Proof of Theorem 3.4.2 Let s_i and s_j be any two tuples in such a stream with $s_i.B \geq s_j.B$. Also let $VI^A(s_i)$ denote the Virtual Index of tuple s_i when ordering is defined on column A . According to Definition 3.4.1 and Definition 3.2.2, we have

$$VI_{max}^A(s_j) - DSF_{A \rightarrow B} \leq VI_{min}^A(s_i) \quad (3.12)$$

Now given stream SF on column A is k_A , to consider the worst case scenario where ordering defined on column B is maximally scrambled, tuple s_i should be positioned as early as possible (i.e. minimize $PI(s_i)$) and s_j should be positioned as late as possible (i.e. maximize $PI(s_j)$) in the stream (recall

$s_i.B \geq s_j.B$). According to Inequality 3.4b in Theorem 3.2.1, we have

$$PI(s_i) \geq VI_{max}^A(s_i) - k_A \quad (3.13)$$

The minimum value of $PI(s_i)$ is obtained when 1) the equality in Equation 3.13 holds and 2) $VI_{max}^A(s_i)$ obtains its minimum value, which is $VI_{min}^A(s_i)$. This happens when there is no tuples with duplicate value as $s_i.A$ on column A . Therefore, the minimum possible value for $PI(s_i)$ is

$$PI(s_i) = VI_{min}^A(s_i) - k_A \quad (3.14)$$

Similarly, according to Inequality 3.4a in Theorem 3.2.1

$$PI(s_j) \leq VI_{min}^A(s_j) + k_A \quad (3.15)$$

The maximum value of $PI(s_i)$ is obtained when 1) the equality in Equation 3.15 holds and 2) $VI_{min}^A(s_j) = VI_{max}^A(s_j)$ in this particular scenario. Therefore, the maximum possible value for $PI(s_j)$ is

$$PI(s_j) = VI_{max}^A(s_j) + k_A \quad (3.16)$$

Substituting both 3.14 and 3.16 into 3.12, we obtain

$$PI(s_j) - PI(s_i) \leq 2 \cdot k_A + DSF_{A \rightarrow B} \quad (3.17)$$

This proves the SF defined over attribute B is bounded by $2 \cdot k_A + DSF_{A \rightarrow B}$. \square

As a side note, Theorem 3.4.2 in some way can be regarded as the generalization of Theorem 3.4.1 for the SF derivation part because in pipelined join on the same attribute the DSF can be defined as from a column to itself. However, such generalization comes with a price. If we use Theorem 3.4.2 to estimate the SF of the intermediate stream in Figure 3.4, we get a looser upper bound value than that using Theorem 3.4.1. However, this does not mean that the bound given by Theorem 3.4.2 is not tight enough. As shown in the proof, the resultant SF value could be much larger in the worst case.

Finally, let us look back at the original problem of pipelined join on a different join attribute. Still take the query in Figure 3.5 as the example. Now we can first derive the SF w.r.t. column A for the intermediate stream S_4 according to Inequality 3.10 in Theorem 3.4.1. Then as long as the DSF from column $S_2.A$ to $S_2.B$ is defined, SF w.r.t. column B for stream S_4 can be obtained directly using Theorem 3.4.2.

3.5 Memory-Constrained WO-Join

The complete memory characterization discussed in Sections 3.3 and 3.4 enables a query processor to accurately evaluate the real-time memory cost for each join operation so that proper buffer space can be allocated in a dynamic fashion. However, because stream inputs are highly volatile, the required buffer size may vary significantly over time. For example, a data-burst of one input stream can cause a large amount of inter-stream delay, leading to excessive memory overhead in a short time. When this occurs, the required buffer space may easily exceed the physical memory available. Motivated by the observation

that memory overheads are mainly attributed to two factors, scrambled tuple ordering and lag among streams, we propose two techniques (and additionally one hybrid approach) to prevent the system from memory overrun while keeping the output quality. Each technique targets one of the two contributing factors to minimize its impact on memory overhead.

3.5.1 Memory-Sort First Strategy

The Memory-Sort First (MSF) strategy aims to tackle the problem of scrambled tuple ordering. The idea is straightforward: the system performs an in-memory sort on all input streams first before the join. After the sorting, the SF value is reduced to the stream's multiplicity, which is the minimal possible SF value for a given stream. Hence, for an MJoin operator, the space needed to buffer tuples from S_i is reduced from

$$k_i + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j / r_j + L_{S_i \hookrightarrow S_j}(t)\}$$

to

$$Mul(S_i) + \max_{\forall j \in n, j \neq i} \{r_i \cdot Mul(S_j) / r_j + L_{S_i \hookrightarrow S_j}(t)\}$$

However, sorting input tuples itself costs extra buffer as well: an in-memory sort on a stream with $SF = k$ requires extra space to buffer the k most recent tuples. As we shall see later in the experiments, such memory cost may be substantial especially when the query plan only involves a single join operator. Nevertheless, the benefit of MSF becomes apparent when the query plan consists of pipelined join operations. Another good side-effect is that output produced using MSF is also ordered. This may be crucial for applications that

are sensitive to the data order. Although MSF reduces memory overheads, it introduces significant latency in producing the results because an input tuple has to wait until k number of its succeeding tuples have arrived before participating in the join operation.

3.5.2 Disk-Buffer First Strategy

The Disk-Buffer First (DBF) strategy deals with streams that are not synchronized. The intuition is when a stream S_i runs *far* ahead of stream S_j , recently arrived S_i tuples will not immediately contribute to any join results since the matching tuples from S_j have not arrived yet. To save memory, new tuples from S_i can be flushed onto the disk first and then retrieved later when their join counterparts from S_j are about to arrive. Essentially, the DBF strategy reduces the memory overheads by minimizing the *Lag* among input streams. The strategy works as follows: Every time a new tuple from the “lagging” stream (say S_j) arrives, it triggers the query engine to read tuples of the “leading” stream (say S_i) from the disk such that both $L_{S_i \leftrightarrow S_j}(t)$ and $L_{S_j \leftrightarrow S_i}(t)$ are 0 or close to 0. In actual implementation, the system reads slightly more tuples of the “leading” stream from the disk in advance such that $L_{S_i \leftrightarrow S_j}(t) > C$, where C is some predefined runtime parameter. The purpose of doing this is to minimize the possibility that the join to be blocked due to the I/O operations on S_i when S_j ’s data rate is higher than the I/O speed. It is important to note that DBF is only beneficial when inter-stream delays are significant. If the *Lag* is so small that the arrival time difference among matching tuples is less than the time for 1 disk read plus 1 disk write, then the “leading” stream will eventually run

behind the “lagging” stream after the I/O operation. In our implementation, DBF strategy is only triggered when the *Lag* is no less than 3 times of the disk I/O time. According to the cost model, with the DBF strategy, the memory needed to buffer a stream S_i can be decreased from

$$k_i + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j / r_j + L_{S_i \hookrightarrow S_j}(t)\}$$

$$\text{to} \quad k_i + C + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j / r_j\}$$

However, similar to MSF, DBF also introduces extra output latency because of the disk I/O operations.

3.5.3 Hybrid Approach

The MSF and DBF strategies are two complementary approaches. They can be combined to achieve even better memory reduction. The hybrid approach first sorts the recently arrived tuples in the main memory, then flushes the “leading” stream tuples onto the disk. According to the cost model, the buffer space required by using the hybrid strategy can be decreased from

$$k_i + \max_{\forall j \in n, j \neq i} \{r_i \cdot k_j / r_j + L_{S_i \hookrightarrow S_j}(t)\}$$

$$\text{to} \quad Mul(S_i) + C + \max_{\forall j \in n, j \neq i} \{r_i \cdot Mul(S_j) / r_j\}$$

Interestingly, our experiment results show that the memory reduction achieved using the hybrid approach is even larger than the sum of the memory reductions using MSF and DBF individually. The main reason is as follows: In

hybrid approach, Lag is measured only after streams are sorted. While in DBF approach, Lag is measured when there are intra-stream delays presented. For the latter case, the Lag value has to be estimated in a conservative way (i.e. a smaller value) so that disk-buffered tuples from the “leading” stream would be retrieved earlier from the disk to cater for the order scrambling of the counterpart streams. This of course leads to extra memory overheads. On the contrary, in the hybrid approach, tuples from the “leading” stream do not have to be retrieved from the disk earlier because without intra-stream delay the system knows exactly when join counterparts from the other streams arrive. In short, the hybrid approach can achieve super reduction in terms of memory consumption because the DBF strategy over a completely sorted stream performs better than using the same strategy over a partially ordered input.

However, the output latency using Hybrid also suffers the most. Therefore, this strategy is suitable for the situation where memory space is extremely limited while a relatively larger output latency is tolerable.

3.6 Experimental Study

We developed a WO-Join prototype system using Sun JDK 5.0. The system consists of two components: the query executor, and the memory manager. The query executor is similar to the main memory version of the MJoin operator [95], except that the buffer size for each stream is dynamically determined by the memory manager. The memory manager updates the stream characteristics such as data rate, SF , DSF and Lag upon the arrival of a tuple (which could be a batch of individual tuples in a batch processing mode) if these parameters

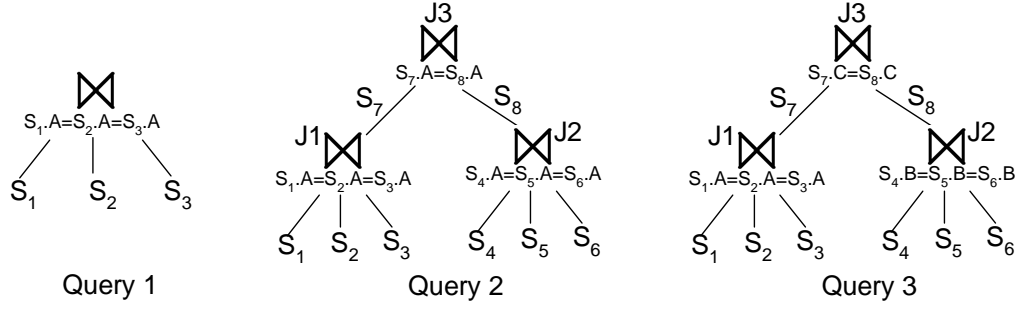


Figure 3.6: Example queries used in this chapter

are not known apriori, and determines the buffer size for the input streams. The prototype system can be easily integrated with a query planner or a query optimizer. Essentially, it takes the query plan graph generated by the planner as the input and then executes the query plan accordingly with the memory management scheme proposed in this chapter.

We ran the experiments on three test queries as shown in Figure 3.6. Query 1 is used to validate our cost model and the memory efficiency of WO-Join, while Query 2 and Query 3 are for evaluating the WO-Join performance over two types of pipelined joins introduced in Section 3.4. A data generator was implemented to produce streams with customizable SF , DSF , multiplicity, Lag and data bursts. By default, streams are generated according to Poisson process with mean inter-arrival time equal to $20ms$. The SF of each stream ranges from 0 to 500. For Query 3, the value of DSF between related columns is set between 0 and 300. The multiplicity of each stream is 2 unless otherwise specified. The size of each tuple is 24 bytes. The experiments were conducted on an IBM x255 server running Linux with four Intel Xeon MP 3.00GHz/400MHz processors and 18G DDR main memory. All experiments were repeated thirty times and the average values were reported. We also varied the above parameters and found

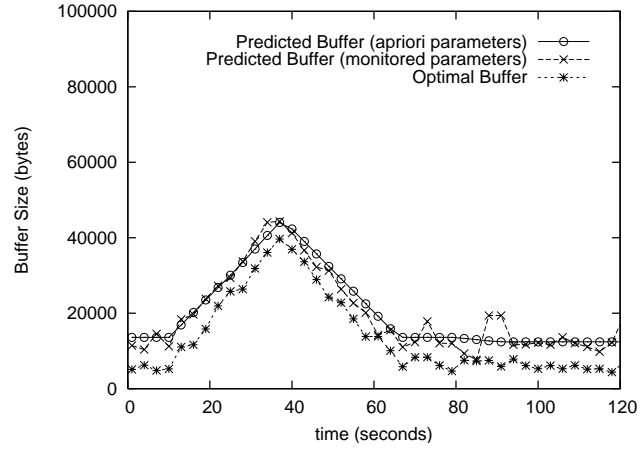
that our conclusions are not sensitive to their values.

Notably, other than the query executor, the memory manager also introduces memory overheads to the system since estimating the stream characteristics, such as the SF , requires maintaining stream states. Here we adopt an idea similar to *k-Mon* [15], which integrates the monitoring mechanism into the query executor to avoid duplicating stream states. Therefore, its overhead is almost negligible.

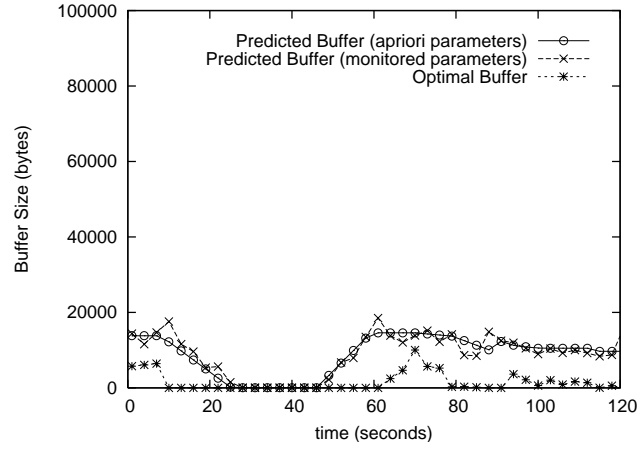
3.6.1 Experimental Evaluation

Memory Cost Model Validation The first experiment aims to validate our memory cost model. We start by running Query 1 and report the maximum buffer size for each input stream predicted by the cost model with and without apriori knowledge about the data characteristics. To compare, for each output tuple produced, we *backtrack* and find out the minimum buffer size required to produce that tuple, and report the maximum values recorded in each second. We call this value the optimal buffer space required to generate the complete join results. At $t = 10s$, we suppress S_2 's data rate to $1/8$ of its initial value to simulate congestion at its source's side. The problem lasts for 30 seconds and then is restored. Similarly, to simulate data bursts, we increase S_3 's data rate by 8 times at $t = 80s$ and hold it for 10 seconds before restoring its initial rate. Figure 3.7 depicts the buffer consumption for the three streams over a period of time. It is clear that the curves for the predicted buffer consumption closely follow the one for the optimal buffer consumption. As we can see, during the period when S_2 's source is congested, the buffer spaces for S_1 and S_3 increase

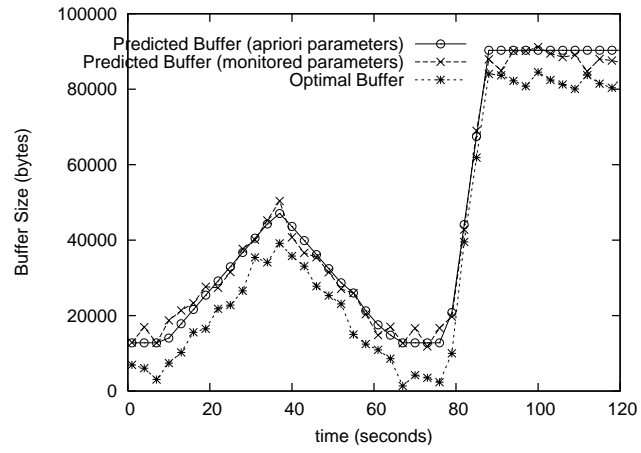
3.6. EXPERIMENTAL STUDY



(a) Stream S_1



(b) Stream S_2



(c) Stream S_3

Figure 3.7: Buffer space for streams in Query 1.

significantly. This is due to the sharp increase of the *Lag* from S_1 to S_2 and from S_3 to S_2 . For the same reason, when there is a data burst on S_3 at $t = 80$, the buffer space for S_3 increases substantially. In either case, the predicted buffer consumption (either based on apriori knowledge or based on parameters monitored during runtime) adapts well with respect to the input variations.

In terms of output quality, when the input parameters (i.e. data rate, SF , multiplicity and *Lag*) are known apriori, the system generates 100% join results. This validates our memory cost model which guarantees complete results given the accurate parameter settings. On the other hand, when the input characteristics are not known beforehand, they will be monitored during runtime by the memory manager for predicting the buffer size. In this case, we obtain an output accuracy of 99.6%. Only minor result tuples are missed even when there is a major change in some of the input parameters which the memory manager does not catch up with instantly. Nevertheless, we can see from Figure 3.7 that the curve for the predicted buffer size based on monitored parameters almost coincides with the one based on apriori input knowledge except for some minor variations. For clarity, only the curve based on monitored parameters will be plotted for the subsequent experiments since it reflects a more realistic scenario. The output accuracy achieved based on monitored parameters is between 99.2% and 99.6% throughout all our experiments.

Next, we perform the similar experiments on Query 2 and Query 3, respectively. We would like to see whether the non-leaf operator (i.e., $J3$ in both queries) adapts well using WO-Join. Similar to the previous experiment, we suppress $J1$'s output rate which causes S_7 to run behind S_8 . Changes in the buffer consumption for S_7 and S_8 are shown in Figures 3.8 and 3.9 for Query 2

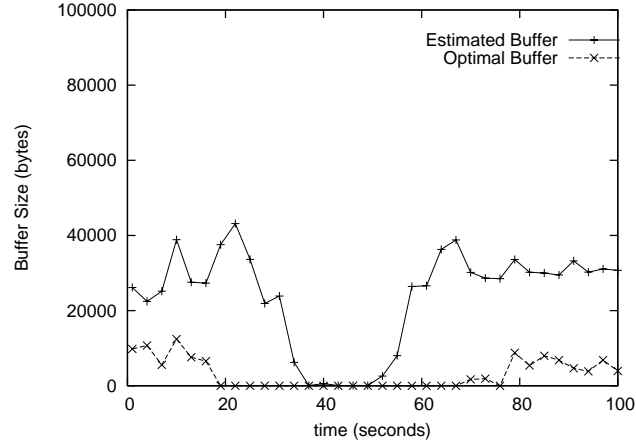


Figure 3.8: Buffer for S_7 (Query 2)

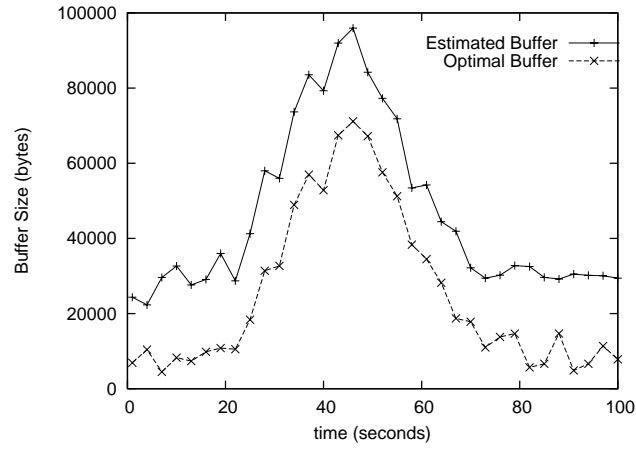
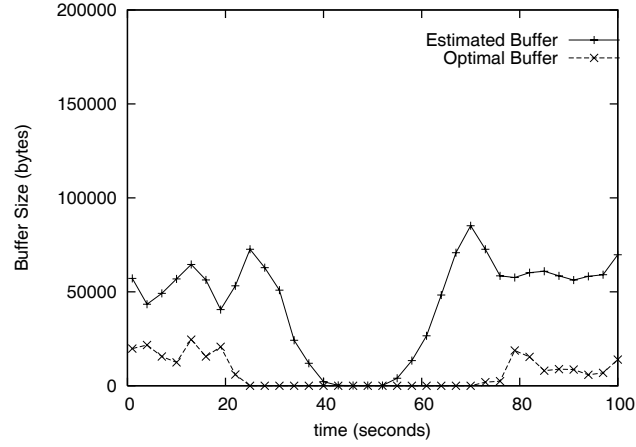
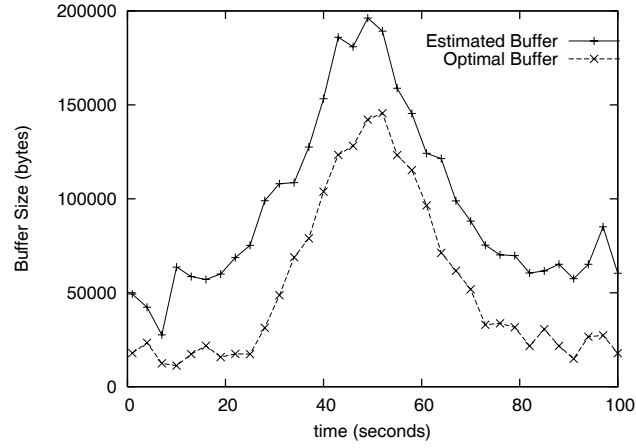


Figure 3.9: Buffer for S_8 (Query 2)

and in Figures 3.10 and 3.11 for Query 3. We observe that this time the margin between the estimated value and the optimal value is much wider compared to the case for Query 1. This is because, for a non-leaf operator, the SF and data rate of the input streams are predicted based on the upper bounds of the statistics of the raw input streams (recall Theorem 3.4.1 and 3.4.2). However, this does not mean the system over-estimates the buffer consumption because the actual buffer required may still hit the predicted value in the worst case.


Figure 3.10: Buffer for S_7 (Query 3)

Figure 3.11: Buffer for S_8 (Query 3)

Memory Overhead Comparison Our next experiment compares the average memory consumption between WO-Join and the traditional W-Join. The experiments on W-Join were conducted as follows: we first set the window size (or the buffer size) on par with the amount of buffer consumed by WO-Join. We achieve an output accuracy less than 40% for all three test queries. Then we slowly increase the window size until it is barely large enough to produce the equivalent output quality as WO-Join (i.e., between 99.2% and 99.6% ac-

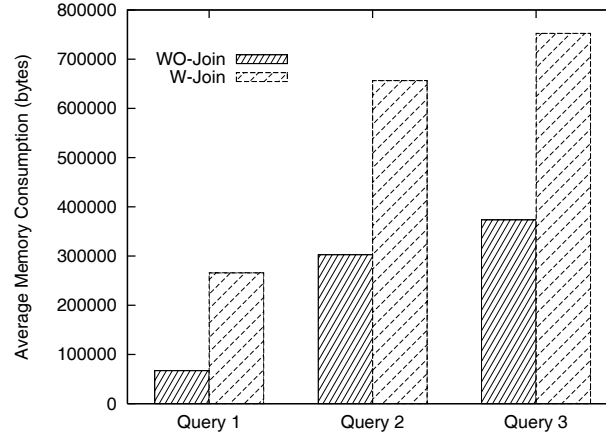


Figure 3.12: WO-Join vs W-Join

curacy). The required memory consumptions are shown in Figure 3.12. As expected, since W-Join fixes the window size throughout the query execution, the required buffer size has to be much larger than WO-Join in order to achieve the same level of output quality.

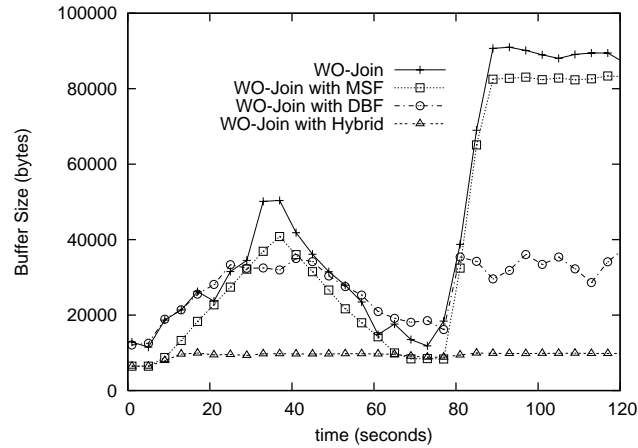


Figure 3.13: Memory reduction strategies

Memory Constrained WO-Join The next experiment evaluates the memory reduction strategies proposed in Section 3.5 using Query 1. Figure 3.13 compares the buffer consumption for S_3 using these strategies (the observa-

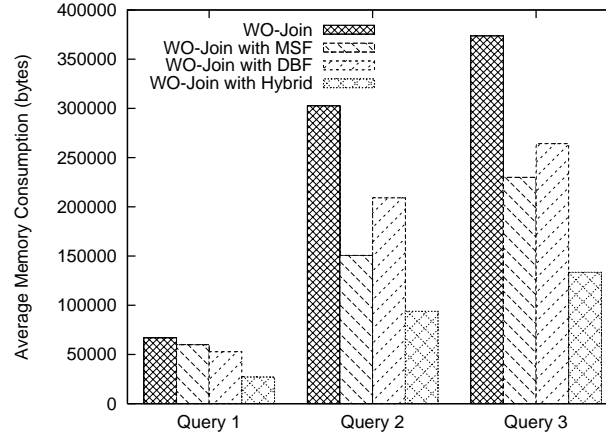


Figure 3.14: Average memory consumption

tions of the buffer spaces for other streams are similar). As shown, the memory reduction achieved by MSF is insignificant. This is mainly because MSF requires additional buffer space for pre-sorting tuples, which almost offsets the savings brought by the ordered sequence. On the other hand, the DBF approach is more effective in this case. This can be attributed to DBF’s ability to “re-synchronize” the streams without much memory overhead. We can see from the figure the buffer size under DBF is always below a certain value regardless of the amount of *Lag* among streams, which is determined by other factors such as *SF* and data rate. It is not surprising that the hybrid approach yields the best memory reduction. As explained in Section 3.5.3, its performance can be even better than the sum of the memory reduction brought by MSF and DBF strategies individually. Since tuples are ordered and synchronized when they are joined, the buffer required at each stream is indeed minimized. Actually, the major memory cost for the hybrid approach comes from the buffer for tuple pre-sorting required by MSF.

Figure 3.14 depicts the average memory consumptions of Queries 1, 2 and 3

under various strategies. Obviously, Hybrid achieves the lowest memory overhead. The interesting observation here is that in Query 1, DBF consumes less memory than MSF, while in both Queries 2 and 3, the reverse is true. There are two reasons. First, for Queries 2 and 3, tuple pre-sorting of MSF is only required at the leaf join operator. Hence, with pipelined join operators, the memory overhead for sorting input streams becomes less significant. Second, the DBF approach suffers from the rough estimation of the upper bound of SF and data rates for intermediate result streams. Therefore, the memory cost is still quite high even though the *Lag* among streams has been minimized.

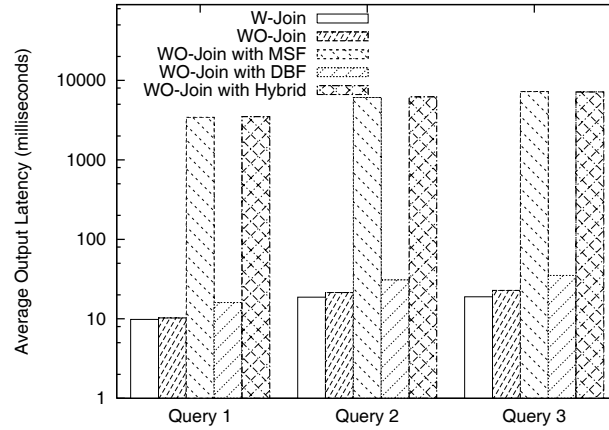


Figure 3.15: Output latencies

Output Latency Lastly, Figure 3.15 compares the average output latencies among W-Join, the normal WO-Join and various WO-Join memory reduction strategies. The latency is measured by the difference between the output time of a result tuple and the arrival time of its last contributing input tuple. We can see that the latency of WO-Join is almost as low as the W-Join approach. This means the computation overhead incurred by WO-Join is negligible. The latency of DBF is about 50% higher than the plain WO-Join due to the I/O

operations involved. The most stunning fact is that the latency of MSF and Hybrid is two orders of magnitude higher than that of WO-Join. The reason is, in MSF or Hybrid, tuples cannot be used to join with other streams upon arrival. Instead, they are stored in the sorting buffer first. The delay introduced by the sorting is much higher than the one caused by the join itself. Therefore, MSF and Hybrid suffer from severe output latencies in this case.

3.7 Related Work

While a plethora of cost-efficient algorithms have been proposed for stream query processing, such as [13, 43, 94], works on memory-efficient processing are relatively fewer. Arasu et al. [7] classifies a broad range of queries into two classes: those can be evaluated with bounded memory and those cannot. Cammert et al. [23] proposes an adaptive memory management approach for W-Join. Optimizing memory consumption has also been studied by Babcock et al. [11]. Different from ours, they approach the issue from the operator scheduling perspective. Also, various stream properties such as the ordered constraint [15] and *slack* [3] are identified in the literature. Some of the ideas are similar to our notion of *SF*. However, they are applied in the context of W-Join solely for the purpose of minimizing memory overheads. We view these stream properties from a different angle. That is, since they (intra/inter-stream delay, etc.) are the crucial factors that contribute to the memory consumption, we may build a memory cost model based on these factors so that stream join can be evaluated in a window-oblivious fashion and high quality results are attainable even in a memory-constrained situation. To the best of our knowl-

edge, this chapter is the first one that proposes the idea of data-driven memory management. Punctuation [37, 91] or heartbeat [53, 85] offers an alternative solution to join stream data without explicit window semantic. However, their work makes different assumptions from ours which relies on the data sources to generate punctuation messages at an appropriate rate. Our proposed strategy does not impose such a requirement.

Works on memory-constrained stream join usually focus on certain metric (such as “Max-Subset” or some other criteria) [5, 33, 60, 86, 104] to obtain a statistically optimal solution. Different from these approaches, our proposed memory-constrained join strategies (MSF and DBF) are based on the understanding of the memory consumption patterns to produce complete or near-complete query answers given the input statistics. The disk-based approach for joining stream data is used in [64, 92, 95]. Unlike these techniques, the DBF strategy aims to minimize the *Lag* among streams and therefore tuples stored in the disk would not match tuples from other streams that are currently buffered in the memory. This means in DBF, the join is less likely to be blocked due to disk I/Os – an important difference from other disk-based join strategies.

3.8 Summary

One important contribution of this chapter is that it answers a fundamental question in stream join processing, that is, exactly how much memory is needed to perform a complete join among multiple (totally or partially ordered) data streams. Based on our study, we contend that a data-driven memory management strategy should be used in place of the traditional query-driven scheme for

many stream join applications whenever feasible. In data-driven memory management, query users are oblivious to the window semantic. The system can dynamically adjust the state buffer size in order to produce quality join results. We studied the memory consumption patterns and identified that intra-stream delay and inter-stream delay are the two main causes for excessive memory utilizations. Based on these observations, we derived the memory cost model and proposed different memory overhead reduction strategies. Our experimental study has demonstrated the effectiveness of our proposed techniques.

4

Tuple-based Data Stream Scheduling

Live information carried by streaming data is often meaningful only when it has been processed and consumed within a certain short time frame, which makes real-time scheduling crucial for stream query processing. This chapter discusses several interesting stream scheduling strategies. we first give the introduction and motivate the problem in Section 4.1 and 4.2. Next in Sections 4.3 and 4.4, we formally define the problem and transform it into a job scheduling issue. Section 4.5 discusses the applicability of data stream scheduling in general. Sections 4.6 and 4.7 introduce our novel scheduling strategies inspired by the job scheduling model. Section 4.8 considers minimizing scheduling overhead through tuple batching. Experimental evaluation of the proposed strategies and the related discussion are given in Sections 4.9 and 4.10. Finally, Section 4.11 concludes the chapter.

	OBS	TBS
problem size	related to # operators	related to # input tuples
parameters	operator-related parameters only	operator-related parameters and tuple values
optimization goal	system-oriented metric	mainly user-oriented metric

Table 4.1: Comparison between OBS and TBS

4.1 Introduction

Typical applications of Data Stream Management System (DSMS) often involve time-critical tasks such as disaster early-warning, network monitoring, and on-line financial analysis. In these applications, timeliness of output delivery is extremely crucial, which sometimes marks the difference between success and failure. Owing to the volatile input characteristics and the ever-changing query environment, to efficiently allocate resources so that the output can be consistently delivered in a timely manner has always been a challenging task.

Existing literature on this issue usually considers it as just another Operator-Based Scheduling (OBS) problem, which is believed to have been well addressed over the past few years. An OBS problem can be briefly described as follows: Given a query graph, intelligently schedule *query operators* such that certain objective is achieved. However, on-time delivery of output tuples requires resource allocation control at tuple level, which an operator-based scheduling is unable to offer. This calls for Tuple-Based Scheduling (TBS) to be implemented in place. A TBS problem is the following: Given a query graph, intelligently schedule *input tuples* such that certain objective is achieved.

OBS and TBS are two fundamentally different types of scheduling problems

though they may seem similar at the first look. In OBS, the question to ask is “which operator to execute next?”. In TBS, however, the question is about “which tuple to process next?”. Their difference also lies in the basis where the scheduling decision is reached. For OBS, the decision is solely based on the properties of the involved query operators, such as execution cost, memory overhead, and selectivity. For TBS, the decision not only depends on the operators’ properties, but also on the content of each input tuple. As in data stream applications the number of input tuples is much larger than the number of query operators (for unbounded data stream, the former is infinity), TBS has a much larger problem size and hence is often harder to solve than the corresponding OBS. The main differences between the two are summarized in Table 4.1.

It is worth noting that the majority of data stream scheduling techniques proposed by the database community so far belong to OBS. Take the Chain algorithm [11] as an example. Its goal is to minimize the total number of tuples buffered in the memory, which is a system-oriented metric. Also, the algorithm, at each scheduling time, chooses an operator for execution from all those with a non-empty input queue by their potential capabilities of reducing the number of tuples. The size of the optimization problem is the number of the operators.

However, there are a number of scenarios in real-time stream applications that can only be addressed using TBS. For example, in an environmental monitoring system that offers disaster early-warning service through the analysis of multiple ecological measures, it is important to ensure the output from the analysis queries is produced in time to avoid critical alert message being delayed. This requires scheduling decision to be made based on individual tuple’s

content (e.g. timestamp). Since OBS does not take tuple's value into account, it is not applicable here. Generally, OBS can only solve scheduling problems related to optimizing system-oriented metrics, such as execution cost or total memory overhead. For problems on optimizing query- or user-oriented metrics, they can only be addressed by TBS. A typical example of a user-oriented metric is QoS which, in the context of data stream applications, is often defined as the maximal tolerable delay between input generation ¹ and the corresponding output production.

The work described in this chapter can be viewed as our initial effort towards tuple-based real-time data stream scheduling. We use QoS (in terms of output latency) as the performance metric for our case study. For each input tuple, we define a validity time window within which the tuple is deemed fresh and meaningful. An output tuple is said to be delivered on time only if all its contributing input remains fresh by the time it is produced. Given that each input tuple is attached with a timestamp indicating when it is generated, the expiry time of an input tuple can be determined by adding the input timestamp with the validity period. The ultimate goal is to efficiently schedule tuples for different query operations so as to produce as many valid output tuples as possible before the input tuples get expired. Although we choose output latency as the performance metric, scheduling strategies discussed in this chapter can be easily extended to other TBS problems.

While a plethora of work on OBS can be found in the literatures [11, 52, 82], there is a dearth of real TBS strategies being proposed. The only TBS strategy

¹Input generation refers to the time when the input is generated at the data source. Readers should not confuse it with input arrival time which denotes the time when the input arrives at the system input queue.

we are aware of is the one proposed by Carney et al. in the Aurora project [25]. However, they only offered a heuristic without much in-depth analysis. In this work, we systematically study this issue from a different angle. By making an analogy between tuple-based scheduling and classic job scheduling, we are inspired to discover a set of novel techniques that bring new insights to the issue. Particularly, we point out that the Aurora approach belongs to one type of greedy strategies, which only covers part of the solution space. With a comprehensive understanding of the issue, our proposed new algorithms offer higher scheduling accuracy and better responsiveness to input load variation.

Important contributions of this work include:

1. Identification of TBS as an important class of stream scheduling problems,
2. An in-depth analysis of how a TBS problem can be transformed into a job scheduling problem,
3. Presentation of two general approaches to data stream scheduling, namely greedy strategy and deadline-aware strategy. Within each approach, two algorithms are proposed with the aim to improve the overall performance from a job scheduling perspective,
4. Experimental studies that identify factors influencing the effectiveness of scheduling strategies and compare the performance of the various solutions.

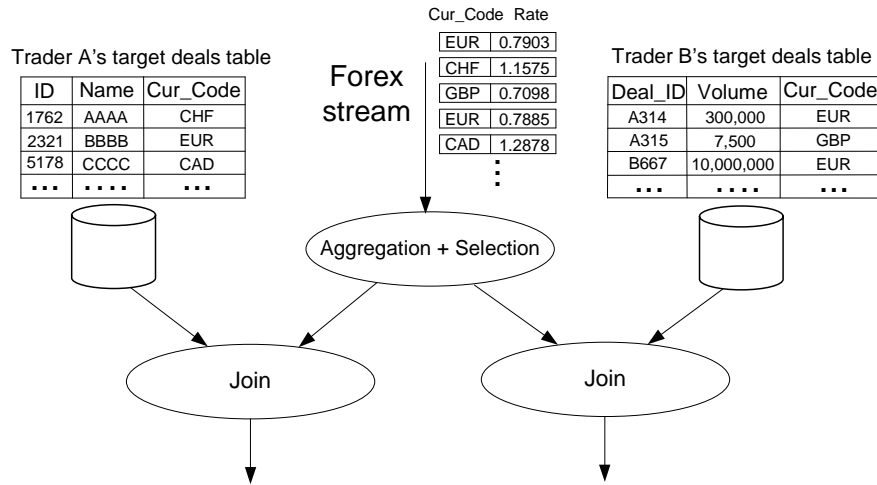


Figure 4.1: Example query in foreign exchange trading

4.2 Motivation and Challenges

4.2.1 Motivating Example

Consider the following scenario in foreign exchange (Forex) trading. People doing business in global markets (such as the global commodity markets) often involve speculation on Forex rate movements when they want to purchase goods traded in different currencies. To minimize the risk due to currency fluctuation, they choose to trigger the trade only when the Forex rates become favorable. What they do is to issue a continuous query to check the real-time Forex rates and clinch the deals as soon as the latest rate meets certain criteria (e.g. falling below a moving average). There are a lot of such traders in the market and Figure 4.1 shows an example where the Forex stream is shared by queries from two traders. Each trader has a list of potential deals (in various currencies) he plans to make. These deal information is stored in his own target deals table. Upon receiving a tuple from the Forex ticker, the query processor first

4.2. MOTIVATION AND CHALLENGES

updates the relevant statistics (such as moving average) in response to the new received input. If the new rate meets the selection criteria (e.g. the rate falls below the 3-hour running average), the tuple will be passed down to join with the target deals tables to populate deals that should take place immediately. Imagine there are hundreds of such traders, each with hundreds of similar deals to make. To ensure queries' prompt responses to maintain a high overall QoS is not easy. Actual trading queries can be much more complex than what is shown in this example and may impose very stringent timing requirements. This is because market data fluctuates rapidly. Real-time Forex information typically expires in a few seconds. To ensure deals are sealed successfully at the desired rate, query results have to be produced while the current rate remains valid. Any delay during query processing that leads to the expiration of the current rate may directly amount to a huge loss.

4.2.2 Challenges

This is a typical scenario where an efficient TBS strategy is desired to ensure the timeliness of output delivery. Given the input generation time and its validity period, one can tell by when all the queries that involve processing this input should be computed. To design a query engine that allocates resource on a per-tuple basis, however, faces two main challenges: The first comes from the inherent complexity of the queries. Each continuous query may consist of various types of operators (unary, binary or even N-nary operators). In addition, operators may be shared among different queries to avoid redundant work. Because of these, when we model the entire query plan as a graph with

each operator being a node and each input/output stream being an edge in the graph, the size of the search space for an optimal execution sequence becomes exponential. Consequently, the benefit brought by a scheduled execution plan may not justify the cost of computing it. The second challenge comes from the data input, whose implication is twofold. Firstly, enormous amount of data input prohibits fine-grained control over resource allocation at individual tuple level. Secondly, the uncertainties about input characteristics such as data rates and arrival patterns place the scheduling issue in a highly volatile environment. These uncertainties inevitably influence the scheduling performance. A good scheduler hence has to be robust enough to cope with such a dynamic setting.

4.3 Preliminaries

4.3.1 Metric Definition

As mentioned before, we adopt the notion of QoS to evaluate the performance of an execution strategy. At application level, an output tuple is considered valuable only if it is produced before any of its contributing input gets expired. Formally, this can be modeled by the utility function below:

Definition 4.3.1 *Given a query Q , let t^γ denote the time when an output tuple γ is produced. And let T_i^γ and L_i^γ be the timestamp and the validity period respectively for some input tuple i that contributes to the output tuple γ . Then the utility function for tuple γ is:*

$$U_\gamma = \begin{cases} 1 & \text{if } t^\gamma \leq \min\{T_i^\gamma + L_i^\gamma\} \text{ for all input tuples} \\ & \text{that contribute to } \gamma\} \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

Accordingly, the query level utility can be evaluated by taking the normalized aggregation of the tuple level utility:

$$\frac{\sum_{\gamma=1}^n U_\gamma}{n}, \text{ } n \text{ is the total number of output tuples} \quad (4.2)$$

In a multi-query environment, we seek to achieve high output quality across all the participating queries. Each query q_i is assumed to be associated with a weight w_i to indicate its importance. A higher w_i implies a higher value of q_i . Now, the utility function U over all the queries is the weighted sum of those individual queries. That is:

$$U = \sum_{j=1}^m \frac{w_j \sum_{\gamma=1}^{n_j} U_\gamma}{n_j} \quad (4.3)$$

where m is the number of participating queries.

For unbounded input streams, the function above should be defined within an observation period. Then the parameter n in the equation refers to the total number of output tuples produced in the recent observation period (say last 5 hours). Note the length of the period does not affect our algorithm. It should be meaningful to the application. For example, if its length is 5 hours, then our algorithm is optimizing the objective function defined on the last 5 hours.

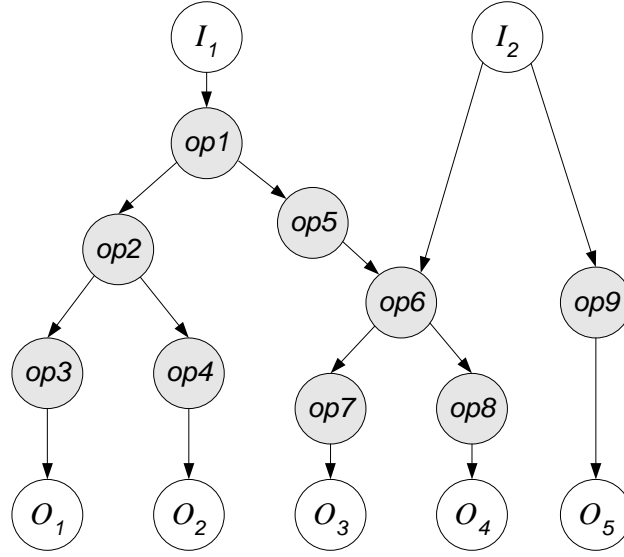


Figure 4.2: A query graph example

4.3.2 System Model

Similar to existing work on stream processing, we model the entire Continuous Query (CQ) plan as a set of Directed Acyclic Graphs (DAG). Vertices with only outgoing edges represent input streams and those with only incoming edges represent output streams. Other vertices are query operators. Edges connecting vertices are tuple queues that link the adjacent operators. Data flows are indicated by arrows. For example, Figure 4.2 shows a query graph with two input streams (I_1, I_2) and five output streams (O_1, O_2, \dots, O_5). Each output stream corresponds to exactly one registered query in the system (O_1 is the output for query Q_1 , and O_2 for query Q_2 ... so on and so forth). Also there are nine query operators ($op1, op2, \dots, op9$) in this plan. Some operators are dedicated to a single query (such as $op3$ for query Q_1) while others are shared among several queries (such as $op6$ for query Q_3 and Q_4). We will use

this query graph as a running example throughout the chapter.

In this problem setting, we assume complete and ordered query results are desired. For each input stream, tuple arrivals are ordered by their timestamps. Each query operator can only process tuples from its input queue in a First-Come First-Served (FCFS) manner so that the tuple order is preserved throughout the query execution.

We also assume each input stream has a predefined validity period. This, plus the tuple creation time (indicated by the timestamp), determines when an input tuple expires. The validity period is a constant, whose value depends on the nature of the data source. For example, tuples from stock tickers may have a short validity period of a few seconds only while data pertaining to temperature readings could usually have a longer lifespan.

4.3.3 Problem Statement

The problem we try to solve can be formalized as follows: *Given the query operator graph, continuously allocate a time slot for some operator to process a tuple presented at its input queue such that the objective function U , defined in Equation 4.3, is maximized.*

4.3.4 Related Work on Data Stream Scheduling

The topic of data stream scheduling has been studied with different objectives. For example, the Chain algorithm [11] schedules query operators in a way such that runtime memory overhead is minimized. In Urhan's rate-based scheduling [93], the objective is to maximize the output rate at the early stage of query

execution. There are also scheduling algorithms proposed for optimizing query response time [52] or its variant metric (such as slowdown in [82]). Note these objectives are all related to system-oriented metrics. Hence, appropriate OBS strategies as proposed in those papers would be sufficient to tackle the problems. The Eddies project [10] is an exception in the sense that it offers highly adaptive query optimization by reordering query operators on a tuple-by-tuple basis. Clearly, this is not an OBS approach. However it is not a TBS either because at each scheduling point, the question to ask is still “which operator to execute next” (or more precisely, “which operator queue should the tuple be put into next”), not “which tuple to process next”. Also, their ultimate goal is to improve the throughput, yet another system-oriented metric. In contrast, we adopt a QoS-oriented view, which brings in the user requirements as another dimension of the issue. Such a slight difference, however, renders totally different problem settings.

Probably the most relevant work to ours is the data stream scheduler proposed for the Aurora project [25]. Although they model the problem in a different way, both works essentially deal with the same issue. However, the way we view and approach the problem distinguishes our work from theirs. In our previous work [102], we have attempted to address the issue from a job scheduling perspective. But it only offered a heuristic that works well under an optimistic assumption. In this chapter, we reexamine the problem and systematically discuss TBS solutions from a broader scope. In fact, the Aurora scheduling can be seen as a primitive type of greedy strategy among all the scheduling strategies discussed in this chapter. Greater details on this can be found in Section 4.6.1.

4.4. FROM STREAM SCHEDULING TO JOB SCHEDULING

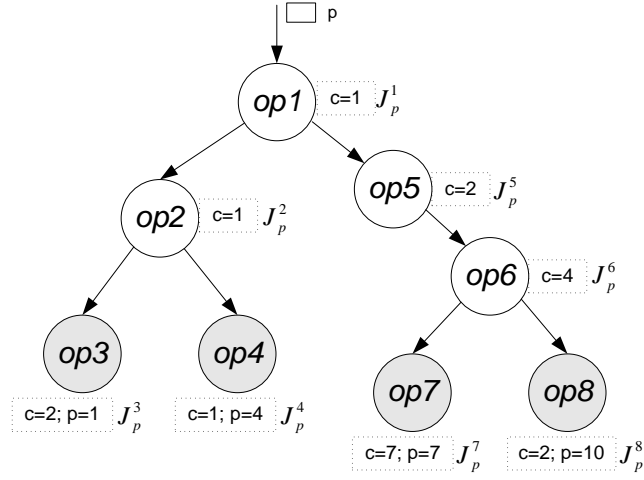


Figure 4.3: From stream scheduling to job scheduling (Input I_1)

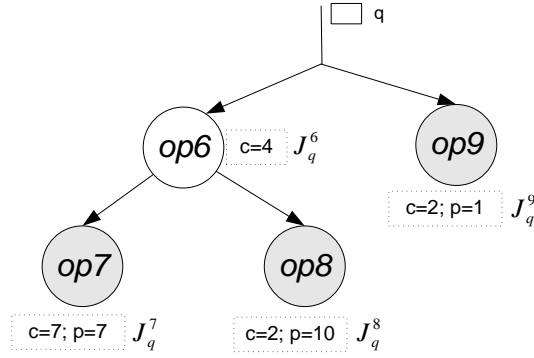


Figure 4.4: From stream scheduling to job scheduling (Input I_2)

4.4 From Stream Scheduling to Job Scheduling

Our first step to approach the issue is to translate a stream scheduling problem into a job scheduling model. Such translation offers a new angle of vision and allows us to analyze the problem from a new perspective.

In a typical single machine job scheduling problem, people look for a plan that allocates each job the appropriate time slot for execution so that the objective function is optimized. Each participating job J_i is associated with a

4.4. FROM STREAM SCHEDULING TO JOB SCHEDULING

processing cost c_i , a deadline d_i and a profit u_i . If J_i can be completed *in time*, i.e. the completion time $t_i \leq d_i$, the profit u_i will be credited; otherwise, the job is considered *late* and no profit will be earned.

Analogously, in continuous query processing, we can treat the work done by a query operator in response to the arrival of a new input tuple as a job. Take the query graph in Figure 4.2 as an example. The arrival of a tuple $p \in I_1$, will eventually trigger eight jobs to be created in the system. Each corresponds to one involved operator. To focus on the essence, we replot the query graph by removing vertices corresponding to input and output streams since they do not participate in query processing. Hence we are left with a pure query plan tree as shown in Figure 4.3. The eight jobs are denoted as J_p^1 to J_p^8 , where J_p^x corresponds to the job performed by operator opx due to the arrival of input p . Similarly, for input stream I_2 the arrival of a tuple q will trigger four jobs to be created for $op6$, $op7$, $op8$ and $op9$, respectively, as shown in Figure 4.4.

4.4.1 Job Cost, Due Date and Utility

In a job scheduling model, each job is characterized by its cost, due date (or deadline) and utility (or profit). In our case, the cost of each job is the product of two parameters: the (average) unit processing cost and the cardinality of the input size. Unit processing cost is the time taken for the operator to process one tuple from its input queue. Cardinality of the input size is determined by the multiplicity (or selectivity) of all the upstream operators along the path from the input stream node to the current operator. For the example in Figure 4.3, Let $m1$ and $m5$ denote the multiplicity of $op1$ and $op5$ respectively. Then the

4.4. FROM STREAM SCHEDULING TO JOB SCHEDULING

input cardinality for $op6$ is simply $m1 \cdot m5$. If c_6 is the unit processing cost for $op6$, the job cost of J_p^6 would be $m1 \cdot m5 \cdot c_6$.

Unlike the traditional job scheduling problem, not all jobs can be assigned explicit due dates here. Firstly, it is important to distinguish two types of jobs in this context: *Leaf-Job* (L-Job) and *NonLeaf-Job* (NL-Job). L-Job refers to jobs associated with leaf operators that produce the final query answers. Examples of L-Jobs are J_p^3 and J_p^8 in Figure 4.3. Intuitively, for L-Job, its deadline coincides with the time at which the corresponding query output is due to be produced. As mentioned, this due date can be computed by finding the minimal expiry date among all the inputs that contribute to the output tuple. For example, if the timestamp of the input p is 100 and the validity period for stream I_1 is 20, the deadline for J_p^3 would be 120 (which is the expiry date of p since p is the only contributing input for Q_3).

NL-Job refers to jobs performed by non-leaf operators. The output of an NL-Job becomes the input of some other NL-Job or L-Job in a query plan. J_p^1 and J_p^6 in Figure 4.3 are examples of NL-Jobs. Unfortunately, often there is no single definite due date for an NL-Job because its due date is defined on each output tuple which does not directly relate to the completion time of NL-Jobs. We will return to the topic of determining an NL-Job's deadline as we discuss deadline-aware stream scheduling in Section 4.7.

Finally, let us examine a job's utility. Again, computing the utility of an L-Job is straightforward, it essentially corresponds to the profit credited to the duly completed output tuple. For NL-Jobs, there is no utility associated with them since completing an NL-Job does not directly contribute to the overall profit. However, the value of performing an NL-Job is implied by the precedence

constraints between L-Jobs and NL-Jobs as specified in the query graph.

4.4.2 Related Work on Job Scheduling

Before delving into the details of the proposed scheduling strategies, let us first briefly review the existing job scheduling techniques that have been explored in the literature. There are various types of job scheduling problems depending on the different objectives and problem constraints. Among them, *Minimizing weighted number of late jobs* is the most relevant type to ours. Essentially, maximizing the total weighted job utility defined in Equation 4.3 is equivalent to minimizing the weighted number of late jobs. Karp [56] proved that the weighted number of late jobs problem in general, denoted as $1||\sum w_j U_j$, is NP-hard. But it is solvable in pseudopolynomial time [58]. Sahni [77] and Gens et al. [40] also proposed the Fully Polynomial-time Approximation Schemes (FPTAS) for the same problem. Moore [72] showed if all the jobs have equal weights, the optimal solution for the original problem can be obtained in polynomial time. A polynomial algorithm is also available if the processing time and the job weight can all be oppositely ordered [57]. In the more recent work [18, 75], solutions and heuristics were proposed to solve the same class of problem with the condition that job release time is not equal. Moreover, Givan et al. [41] offered an interesting scheduling policy that has the guarantee of achieving no more weighted loss than the corresponding Earliest-Deadline-First (EDF) strategy for any input pattern.

However, data stream scheduling is more complicated than the above classic job scheduling problems because it involves additional constraints and practical

concerns. For example, the above strategies assume jobs are independent. But in our case, jobs are dependent as downstream L-Jobs cannot be started until all their upstream NL-Jobs have been completed. That is why a DAG graph is needed to capture the precedence constraints among jobs. And it can be shown that such a problem, denoted as $1|graph|\sum w_j U_j$, is generally intractable.

4.5 Applicability of Data Stream Scheduling

Before delving into the details of various scheduling strategies, let us first consider the applicability of data stream scheduling algorithms, an issue which has been largely ignored by the previous work. The question is whether scheduling strategies are always beneficial or they only work for certain scenarios. Consider the inevitable overhead associated with each scheduling strategy, it would be pointless if such an algorithm cannot ensure a significant improvement in terms of overall system utility. We notice that there are situations where a query's QoS is beyond the scheduler's control. In this work, we would like to exclude those scenarios and focus on queries on which scheduling algorithms would have a definite positive impact. Such type of queries can be categorized as the following: Each operator in the query graph has to be a causal system, with Predictable Output Delivery (POD). It means upon receiving the input and computing resource, the operator is expected to produce the output, with predictable processing delay. The formal definition is given below.

Definition 4.5.1 *Consider an arbitrary query operator, denoted by O . Let j be any one of O 's input tuples and let k be any one of its output tuples which j contributes to. Given that CPU is allocated to operator O to process j*

immediately after j 's arrival at O 's input queue, O is said to be a POD operator if k is guaranteed to be produced in \mathcal{L} time units, where \mathcal{L} is a constant.

Intuitively, for unary operator, \mathcal{L} is equivalent to the operator processing cost, which is often known or can be easily estimated. For binary or N-nary operator, \mathcal{L} embraces both operator processing cost and delay due to incoordination among matching tuples from different inputs. The latter can be bounded or unbounded. For example, without any assumption such delay in join operation is unbounded since the output will be indefinitely delayed as it is never known when matching tuples from the counterpart input will arrive. Fortunately, we find for many applications such bound often exists by analyzing the application scenario or system characteristics (e.g. through heartbeat [85] or punctuation [91]). Therefore, we see a majority of operations in data stream applications can be safely classified as POD operations. Scheduling strategies discussed in this chapter will focus on POD operations only.

4.6 Greedy Strategies

We first consider using greedy algorithms to approach the stream scheduling issue. Greedy-based strategies have been widely employed in job scheduling as well as various resource allocation schemes. In this section, we first give an overview of a basic greedy implementation in the context of data stream scheduling. Then we refine the solution by proposing a new algorithm for achieving the optimal result.

4.6.1 Basic Strategy

The idea of a greedy strategy is simple: Given that each job has a defined profit and cost, one can compute the profit density (i.e. profit-to-cost ratio) and schedule jobs according to the non-ascending order of their profit density values. In addition, because our problem deals with real-time data streams, we need an online mechanism to facilitate job preemption. That is, whenever a new job is generated with higher profit-to-cost ratio than the current running job, the scheduler should immediately put the running job in suspension and pick the new job for execution.

A simple realization of such a mechanism can be done through an interrupt handler as sketched in Algorithm 1. When a tuple arrives at some operator's input queue, we say a new job has been *substantiated*. The job, which corresponds to the work of processing this tuple by that operator, triggers the *NewJob* interrupt. The *NewJob* interrupt handler first computes the profit density of the new job (or new jobs if the interrupt is triggered by multiple simultaneously substantiated jobs) through the function *CompDensity()*. Then it preempts the current executing job if any of the new jobs is found more profitable.

Algorithm 1 Greedy Strategy

Handler_NewJob:

- 1: Let J denote the set of jobs that triggers the interrupt;
 - 2: $\text{CompDensity}(J)$; /* Compute the profit density for each of the job in J */
 - 3: Insert jobs in J into a priority queue H . H is sorted in non-ascending order of profit density. Ties are broken by choosing the job with an earlier due date.
 - 4: **if** the profit density of the first element in H is greater than the current running job's profit density **then**
 - 5: Preempt the current job and execute the first job in H ;
 - 6: **end if**
-

For example, in Figure 4.2, assume all the operators' input queues are empty initially. Now the arrival of a new input from I_1 causes one job to be created corresponding to the work done by operator $op1$. When this job is done, the output tuple is fed into the input queues of both $op2$ and $op5$. This causes two jobs to be substantiated simultaneously corresponding to the work to be done by these two operators respectively. Such process continues as data are processed in a pipelined manner. Along the way the *NewJob* interrupt is triggered whenever new jobs emerge. It ensures the executor is always processing the job with the highest profit density in the system.

4.6.1.1. The Aurora Scheduler

We use the Aurora Scheduler [25] to illustrate how the greedy strategy is implemented in practice. It is one of the very few tuple-based stream schedulers we are aware of that aim to promote the timeliness of output delivery.

Although Aurora uses another way to model the problem and has a more sophisticated scheduling policy, the underlying principle is no different (i.e. a greedy approach). The notion of “gradient” in Aurora is essentially the profit density in our job scheduling model.

The key component of such a greedy-based scheduler is the valuation of jobs’ profits (implemented by the *CompDensity()* function). Because all NL-Jobs in a job tree do not have explicit profit values, evaluating the profit density for these NL-Jobs becomes knotty. In Aurora, the scheduler simply values the profits of executing a non-leaf operation to be the sum of the profit of all its downstream leaf operations. Clearly, this is a very rough estimation.

Another important issue concerning the implementation of the basic strategy is the overhead of deploying such a *NewJob* interrupt handler. The interrupt needs to be triggered for each new tuple present at an operator’s queue. This is obviously infeasible due to the huge input volume and the considerable cost associated with each invocation of the interrupt handler. Aurora approaches this issue through tuple batching. Tuple batching effectively reduces the total number of interrupts though to a certain extent the scheduling performance may be impaired.

4.6.2 Improving Scheduling Accuracy

As mentioned, the performance of a greedy-based scheduler heavily relies on the function that evaluates jobs’ profits. Due to the complex dependency relationships among jobs, precise evaluation of jobs’ profits becomes difficult. Hence, existing implementation prioritizes jobs according to their estimated profit val-

ues. However, such approximation may seriously jeopardize the scheduling accuracy. In this section, we explore a new strategy which, on top of the perplexing job dependency constraints, accurately selects the most profitable execution plan, leading to the optimal scheduling result.

4.6.2.1. Job Tree for Scheduling

Now let us take a closer look at the jobs to be scheduled. Consider the query in Figure 4.2 as an example. As mentioned, the arrival of a tuple from I_1 will eventually trigger eight jobs to be generated though many of these jobs have not been substantiated at the time when the input arrives due to job dependencies. Nevertheless, we can model these jobs as a tree-structured graph, which we call Input Job Tree (IJT), to capture such dependencies as precedence constraints. Each node in an IJT corresponds to the job of performing a particular query operation for the new input. An IJT structure essentially resembles a subset of the original query graph (refer to Section 4.4 on translating from query tree to job tree). It can be derived by abstracting from the original query graph all the subtrees rooted at the operators that directly consume the new input. For example, Figures 4.3 and 4.4 can also be seen as the IJTs generated due to the arrival of the input from I_1 and I_2 respectively. It is important to note that all these jobs have the same deadline (since they are generated as a result of the same input). Hence, for each IJT, we can rephrase our problem as follows: Given the available processor time (i.e. the time from now to the time when the arrived input gets expired), schedule the new jobs such that the total profit (utility) is maximized.

Within an IJT, L-Jobs' profits are defined as usual, which are equivalent to

$c(D_i)$	processing time of node D_i
$p(D_i)$	profit of node D_i
$\rho(D_i)$	profit density of node D_i , $\rho(D_i) = p(D_i)/c(D_i)$
$Par(D_i)$	parent node of node D_i
$Chd(D_i)$	the set of child nodes of node D_i
$DesL(D_i)$	the set of leaf nodes descendent to node D_i

Table 4.2: Important notations used in the algorithm

the profits earned by delivering the corresponding query output on time. For NL-Jobs, we do not attempt to estimate their profits. All of the NL-Jobs in an IJT entail zero profit. This means their account is completely implied by the job precedence constraints. The cost of a job in an IJT is defined as the processing delay of the corresponding operation in the original query graph. For POD operation, the value is \mathcal{L} (refer to Definition 4.5.1). For non-POD operation, the value is set to the average processing delay of the corresponding operation.

Different from the basic strategy which only schedules substantiated jobs, our improved strategy takes the entire IJT for consideration. Since the initial IJT has included all the jobs eventually triggered by the same input, the new interrupt handler hence only needs to be invoked when there is new input entering the query system (as opposed to one interrupt for each job substantiation in the basic strategy).

4.6.2.2. OptProfit Algorithm

In what follows, we present an improved greedy strategy (Algorithm 2), which employs a new algorithm called *OptProfit* to find the optimal execution sequence for a given IJT. Important notations used in this section are listed in

Table 4.2. Other symbol conventions include the followings: A capital letter without any super- or sub-script usually refers to a set or list. A capital letter with a super- or sub-script refers to an element in a set or list.

Unlike the basic strategy which schedules jobs on per query operation basis, the *OptProfit* algorithm uses per *fragment* based scheduling. Here, a fragment refers to a set of connected vertices with at least one of the vertices being a leaf node of the job tree. A fragment, denoted by \mathcal{F} , can be identified by the root node of the fragment and all the leaf node(s) it includes. For example, in Figure 4.3, the fragment that includes $\{J_p^1, J_p^2, J_p^3\}$ can be written as $\mathcal{F}_{\{J_p^3\}}^{J_p^1}$. As another example, the fragment $\mathcal{F}_{\{J_p^4, J_p^7\}}^{J_p^1}$ refers to $\{J_p^1, J_p^2, J_p^4, J_p^5, J_p^6, J_p^7\}$. Fragment based scheduling has several advantages. Firstly, by merging several jobs (operations) into one fragment, we are able to assign a definite profit value for that fragment while in the basic strategy we are unable to do so for non-leaf operations. Secondly, as it turns out, a fragment based *OptProfit* strategy reduces the complex tree-structured precedence constraints to linear precedence constraints. Such simplification enables clairvoyant planning by scheduling non-substantiated jobs without violating the precedence constraints. Lastly, In fragment based scheduling, multiple operations are treated as a single scheduling unit. This effectively reduces the computation overhead.

For the same reason as in the Aurora strategy, tuples are considered in batches in *OptProfit*. It means the scheduler runs for a train of input tuples. Correspondingly, each job now refers to the work of processing a batch of input (as opposed to a single tuple). Two important parameters in *OptProfit* are job graph G and total available processing time C . Initially G is the IJT, which consists of the jobs triggered by the arriving input. Each job is modeled as a

node in G . There are two types of such nodes: *profit node* and *non-profit node*. A profit node in G corresponds to a job associated with a positive profit and a non-profit node otherwise. According to our system model, all L-Jobs are profit nodes. C is the total available processing time for the current scheduling round, it is essentially the time left until the first tuple in the batch gets expired.

The main idea of the *OptProfit* strategy (Algorithm 3) is as follows: If the current best node (i.e. the one with the highest profit density) happens to be a root node, it can be safely scheduled first because a root node does not depend on any other job node. Otherwise, the situation becomes quite complicated because there is no obvious choice for the node to be scheduled first that would eventually produce the optimal solution. For this reason, *OptProfit* adopts fragment based scheduling. Each fragment is carefully formed by iteratively applying the *NodeMerge()* function (refer to Algorithm 4) that merges a node with its parent. When two nodes merge, it means in the final schedule the two corresponding jobs will be executed one immediately after the other (i.e. the child job is executed immediately after the parent job). This node merge process continues until some root node finally gets merged, which completes a fragment. The obtained fragment is guaranteed to have the highest profit density among all the possible fragments that can be found from the current job graph.

Algorithm 2 Improved Greedy Strategy

Handler_NewInput:

- 1: Let J denote the set of the jobs triggered by the arriving input, including both substantiated and not yet substantiated jobs;
 - 2: $U = \mathbf{OptProfit}(J)$;
 - 3: Insert the fragments in U into a priority queue H . H is sorted in non-ascending order of profit density. Ties are broken by choosing the one with an earlier due date.
 - 4: **if** the profit density of the first element in H is greater than the current running fragment's profit density **then**
 - 5: Preempt the current running fragment and execute the first fragment in H ;
 - 6: **end if**
-

Algorithm 3 OptProfit(J)

```

1: Let  $G$  be the IJT constructed from  $J$ ;
2: Let  $C$  be the available processing time. It is the time left until the first
   tuple in the input batch gets expired;
3: Let  $U$  be the computed schedule, which consists of a sequence of fragments.
   Initially,  $U := \emptyset$ ;
4: Let  $P$  denote the total profit of  $U$ , i.e.  $p(U)$ . Initially,  $P := 0$ ;
5: Let  $D$  be the set of profit nodes in  $G$  sorted in non-ascending order of profit
   density;
6: repeat
7:    $D_k := \text{NodeMerge}()$ ;
8:   if  $D_k$  is not null then
9:     /*  $D_k$  exactly subsumes the next best fragment */
10:     $P := P + p(D_k)$ ;
11:     $c(U) := c(U) + c(D_k)$ ;
12:     $D := D \setminus \{D_k\}$ ; /* remove  $D_k$  from  $D$  */
13:     $G := G \setminus \{D_k\}$ ; /* remove  $D_k$  from  $G$ . If  $\text{Chd}(D_k) \neq \emptyset$ , then each of
       its child node becomes a root node after  $D_k$  is removed from  $G$  */
14:    Enqueue  $D_k$  (as a fragment) into  $U$ ;
15:   end if
16: until ( $c(U) \geq C$ )  $\vee$  ( $D$  is  $\emptyset$ );
17: if  $c(U) > C$  then
18:   Only include a fraction of the last scheduled batch such that  $c(U) == C$ ;
19: end if
20: return  $U$ ;

```

Algorithm 4 NodeMerge()

```

1:  $cur\rho := 0$ ; /*  $cur\rho$  records the current best profit density */
2:  $D_{cand} := null$ ; /* candidate node to be merged */
3: for  $i = 1$  to  $|D|$  /*  $|D|$  denotes the size of  $D$  */ do
4:   if  $\rho(D_i) \leq cur\rho$  then
5:     break; /* no longer need to check the rest  $D_i$  */
6:   else if  $D_i$  is a root node then
7:     if  $\rho(D_i) > cur\rho$  then
8:        $cur\rho := \rho(D_i)$ ;
9:        $D_{cand} := D_i$ ;
10:    end if
11:  else if  $(p(D_i) + p(Par(D_i)))/(c(D_i) + c(Par(D_i))) > cur\rho$  then
12:     $cur\rho := (p(D_i) + p(Par(D_i)))/(c(D_i) + c(Par(D_i)))$ 
13:     $D_{cand} := D_i$ ;
14:  end if
15: end for
16: if  $D_{cand}$  is a root node then
17:   return  $D_{cand}$ ;
18: else
19:   /*  $D_{cand}$  needs to be merged with its parent */
20:   UpdateGraph( $D_{cand}$ );
21:   return  $null$ ;
22: end if

```

Algorithm 5 UpdateGraph(D_{cand})

- 1: Let $D_{cand}^p := Par(D_{cand})$;
 - 2: Update G as follows:
 - (1) Merge D_{cand} and D_{cand}^p into one node (denoted by D_{mg}). D_{mg} embeds the job execution sequence D_{cand}^p followed by D_{cand} ;
 - (2) For all nodes that are previously the children of D_{cand}^p (except D_{cand}), reassign their parent node to D_{mg} ; i.e. $\forall D_j \in Chd(D_{cand}^p), D_j \neq D_{cand}, Par(D_j) := D_{mg}$;
 - 3: Update D as follows:
 - (1) $D := D \setminus \{D_{cand}\}$;
 - (2) **if** D_{cand}^p is a profit node, **then** $D := D \setminus \{D_{cand}^p\}$; **endif**
 - (3) Insert D_{mg} into D ;
-

To ensure optimality, node merge only occurs on the pair of nodes that will have the highest combined profit density. To find such a pair, jobs in D need to be checked one by one with their new profit density value after being merged with its parent. Such checking can terminate when the current best profit density value is found to be greater than or equal to that of any of the remaining jobs in D that have not been checked (lines 4-5 of Algorithm 4). After the merge operation, the job graph G and the priority queue D should be updated accordingly (Algorithm 5).

4.6.2.3. Proof of the Optimality of the OptProfit strategy

We show in this section that the *OptProfit* algorithm produces the optimal job execution sequence.

First, we need to prove the following corollary:

Corollary 1 *All job execution sequences imposed by node merge operations appear in at least one of the optimal schedule plans (if there exists multiple optimal plans).*

Proof: Let D_{cand} and D_{cand}^p denote the pair of the child and parent node to be merged by the $NodeMerge()$ function. And let D_{mg} denote the merged node, which embeds the execution sequence that D_{cand} is executed right after D_{cand}^p . The corollary says there exists an optimal plan in which D_{cand} is also executed right after D_{cand}^p . We can prove this by contradiction. Now assume the optimal execution sequence consists of $.., D_{cand}^p, .., D_x, .., D_{cand}, ..$ where D_x represents either one or several nodes between D_{cand}^p and D_{cand} .

Case 1: $\rho(D_x) < \rho(D_{mg})$

Because lines 3-15 of Algorithm 4 ensure that $\rho(D_{mg})$ has the maximum profit density value among any pair of nodes between a node in D and its parent, hence D_x should be scheduled after D_{cand} unless D_{cand} depends on D_x (i.e. D_x is an ancestor of D_{cand}). However, according to our system model, each node in a job graph can have at most one parent. Given D_{cand}^p is the parent of D_{cand} , D_x has to be an ancestor of D_{cand}^p as well. But this is not possible because in the optimal plan D_{cand}^p is executed before D_x .

Case 2: $\rho(D_x) > \rho(D_{mg})$

Similarly, this implies that D_x depends on D_{cand}^p (i.e. D_{cand}^p is an ancestor of D_x) because otherwise D_x should be executed before D_{cand}^p , which will produce a better plan. Let D_y denote the node who is the child of D_{cand}^p and an ancestor of D_x (D_y and D_x could be the same node). Because of the precedence constraint, D_y should be between D_{cand}^p and D_x in the optimal plan. Now consider the

merged node D_{xy} which includes all the nodes along the path from D_y to D_x . If $\rho(D_{xy}) < \rho(D_{cand})$, then a plan with D_{cand} before D_{xy} produces higher profit. On the other hand, if $\rho(D_{xy}) \geq \rho(D_{cand})$, then it contradicts with the code that ensures D_{cand}^p merged with D_{cand} produces the largest profit density. So the only possibility left is D_y and D_{cand} are the same node. However, this is also impossible because it violates the precedence constraint as in the optimal plan D_x is executed before D_{cand} .

Case 3: $\rho(D_x) = \rho(D_{mg})$

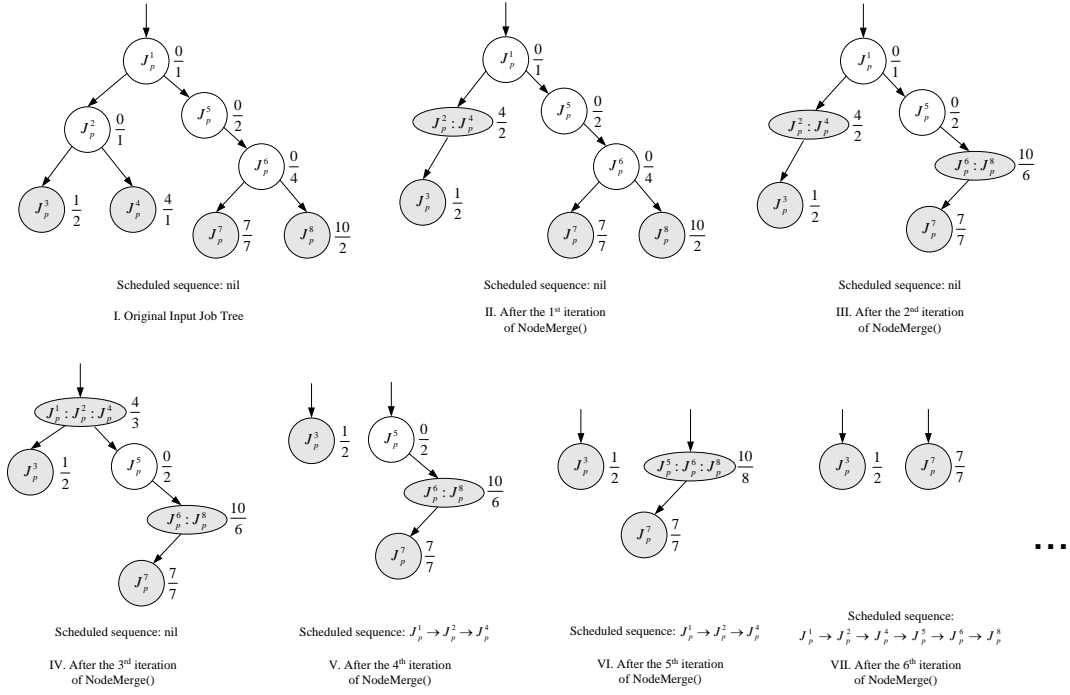
This is possible. However, this does not conflict with the premise because we can remove D_{cand} and insert it right before D_x . In this case, the total overall profit is unaffected. Notice that by moving D_{cand} forward in the schedule here, we do not need to worry about precedence constraint because D_{cand} only depends on D_{cand}^p which is still before D_{cand} .

Cases 1, 2 and 3 complete the proof for Corollary 1. \square

We next prove the optimality of the *OptProfit* strategy.

Theorem 4.6.1 *Given the amount of processing time, the OptProfit algorithm produces the optimal job schedule that maximizes the overall profit.*

Proof: We briefly sketch the proof. The basic idea is by induction on the number of job nodes. Clearly the algorithm is correct when the job graph consists of one job only. When there are n jobs in the job graph, each node merge operation reduces one node in the graph. We have proved in Corollary 1 that each node merge guarantees the execution sequence subsumed by the composite node appears in the optimal plan. And the node merge operations are independent among each other. Eventually, when there are no more node merge occurs,


 Figure 4.5: Illustration of the *OptProfit* algorithm

the problem becomes a knap-sack problem, where each fragment corresponds to an item to be put in the knap-sack. Because we assume each input corresponds to a train of tuples instead of one (and the train size is significantly larger than one). We can take a fraction of the tuples in the last fragment scheduled in the execution plan, just like the fractional knap-sack problem to get the optimal value. \square

4.6.2.4. Illustrative Example

Let's take the query graph in Figure 4.3 as an example to illustrate how the *OptProfit* algorithm works. For simplicity, assume the multiplicities of all the query operators are 1. Now a batch of input tuples p triggers a set of jobs to be created. The IJT is depicted in part I of Figure 4.5. On the right of each

job node, a $\frac{x}{y}$ means the corresponding job costs y units of time and produces x units of profit upon completion in time. Because currently the node with the highest profit density (i.e. J_p^8) is not a root node, *NodeMerge* begins. After the 1st iteration, J_p^2 and J_p^4 are selected to be merged (part II of Figure 4.5). This implies an execution sequence of J_p^2 immediately followed by J_p^4 appears in the optimal schedule. The profit (cost) of the merged node is the sum of the profits (costs) of all its contribution nodes. Parts III and IV of Figure 4.5 show how the job graph evolves after the 2nd and 3rd iteration of *NodeMerge*. At this point, we can see that J_p^1, J_p^2, J_p^4 are merged into one composite node, which is also a root node. In the next iteration of *NodeMerge*, it is found that this composite node has the highest profit density. (The other composite node (J_p^6, J_p^8) has less profit density after it is merged with node J_p^5 .) Hence, the node (J_p^1, J_p^2, J_p^4) , which represents the fragment $\mathcal{F}_{\{J_p^4\}}^{J_p^1}$, is selected into the schedule for execution. The algorithm then continues to run on the rest of the job graph. As can be seen from Parts VI and VII, the composite node (J_p^5, J_p^6, J_p^8) , i.e. $\mathcal{F}_{J_p^8}^{J_p^5}$, is the second fragment selected into the schedule. *OptProfit* terminates when either the job graph G becomes empty or the total available CPU has been exhausted. For example, if the input batch size is 60 and the batch expires in 420 units of time. Then the optimal schedule computed by *OptProfit* is to first execute the jobs in the sequence of J_p^1, J_p^2, J_p^4 till the depletion of the input tuples. This consumes $60 \times 3 = 180$ units of time. Then it schedules J_p^5, J_p^6, J_p^8 for half of the input tuples (i.e. 30 tuples), which consumes the remaining 240 units of time.

4.7 Deadline-Aware Strategies

The greedy strategy looks fine at first glance. However, the experiment shows that it does not often perform satisfactorily. One important reason for this is that the logic behind the greedy strategy, which solely concentrates on maximizing job profits, itself is flawed. Such a strategy lacks a mechanism to enforce the commitment of output generation by its deadline. As stream scheduling deals with live data that generate and expire dynamically, ideally a good scheduler should monitor the lifespan of each input so as to ensure the corresponding results are produced in a timely manner.

The real-time system community has offered abundant tactics that focus on in-time job completion. The representative strategies are Earliest-Deadline-First (EDF) and Least-Laxity-First (LLF). However, these algorithms go to the other extreme in that jobs are prioritized solely according to deadlines (without considering jobs' profits at all). It turns out that EDF and LLF guarantee optimality when the system is underloaded, but perform fairly poorly under overload situations [66]. Unfortunately, online applications including data stream processing are prone to (intermittent) overloading.

In short, what we need is an online scheduler which considers the subtle interplay between job profit and deadline². By online scheduler, it implies the scheduler has no prior knowledge about the jobs (i.e. job's profit, processing cost and due date are not known until the job is present at the input queue). In addition, the produced schedule has to observe certain precedence constraints

²The Aurora scheduler does consider both job profit density and deadline. However, deadlines are only used to break ties when multiple jobs have the same profit density. Consequently, the impact of deadline on job execution sequence is almost negligible.

due to various job dependencies. The ultimate goal is to maximize the total profits of the jobs completed in time, in both underload and overload situations. Needless to say, this is a very challenging task. In fact, it has been shown that no online algorithm exists that guarantees the optimal performance under overload situation. The discouraging result was proved by Baruah et al. [20]. They showed that the best competitive factor any on-line algorithm can achieve, as compared to the optimal clairvoyant scheduler, is no more than $1/(1 + \sqrt{k})^2$, where k is the ratio between the highest and the lowest profit density of the jobs in the system. Therefore, in practice people turn to good heuristics that can work well under a given application scenario.

4.7.1 Deadline-Dominant Strategy

Our new *Deadline-Dominant* (DD) strategy aims to promote the awareness of job deadline in conjunction with profit density. It augments the basic EDF/LLF by introducing job profit density as an important dimension for consideration. One issue concerning deadline-aware stream scheduling is to determine the due dates for NL-Jobs in an IJT. As an NL-Job may be shared by multiple downstream L-Jobs, it has correspondingly multiple due dates, each for one of its L-Jobs. Here, we define an NL-Job's due date to be the one that ensures all its L-Jobs can be completed in time if the NL-Job is committed by that deadline.

The main idea of the DD strategy is as follows: While more urgent tasks are always given a higher priority to execute in the first instance, jobs with significantly higher influence on the overall profit reserve the privilege to preempt other jobs in the future. The strategy can be realized through two interrupt

handlers as sketched in Algorithms 6 and 7. The first *NewJob* interrupt handler (Algorithm 6) is similar to the LLF strategy in that a job with the least (positive) laxity time has the highest priority. The second handler (Algorithm 7) is triggered when some job in the ready queue is going to expire if it is not executed immediately. We call it the *LastDitch* interrupt. The *LastDitch* interrupt essentially gives jobs with significantly higher weights another chance for execution. The parameter u (line 3) sets the threshold that determines when a job's profit density is large enough to preempt the current executing job. Obviously, u must be a value greater than 1. To find an optimal u value is a complicated issue which depends on a few factors (distribution of jobs' profit densities, system workload, etc). We will revisit the issue of how to pick up a good u value through our empirical study in Section 4.9.2.

Note the DD strategy also requires evaluating jobs' profit density values (line 2 of Algorithm 6). The greedy strategies described in the previous section offer two alternative ways to compute profit density: 1) using an approximation method similar to the Aurora's approach and 2) using *OptProfit* algorithm. Both can be seamlessly integrated into the DD strategy. Algorithm 6 actually shows an interrupt handler which integrates the Aurora's approach to compute profit density. If the *OptProfit* is adopted instead, the algorithm will be similar but with the following differences: firstly, the *NewJob* interrupt will be replaced by *NewInput* interrupt; secondly, per job scheduling will be changed to per fragment scheduling.

4.7. DEADLINE-AWARE STRATEGIES

Algorithm 6 The *NewJob* Handler for DD Strategy

Handler_DD_NewJob:

- 1: Let J denote the set of jobs that triggers the interrupt;
 - 2: $\text{CompJob}(J)$; /* Compute the deadline laxity and the profit density for each of the job in J */
 - 3: Remove jobs with negative laxity values from J ;
 - 4: Insert jobs in J into a priority queue H . H is sorted in non-descending order of laxity value. Ties are broken by choosing the job with higher profit density;
 - 5: **if** the laxity of the first element in H is less than the current running job's laxity **then**
 - 6: Preempt the current job and run the first job in H ;
 - 7: **end if**
-

Algorithm 7 The *LastDitch* Handler (for both DD and PD)

Handler_LastDitch:

- 1: Update the profit density of the current executing job (fragment) s . i.e. its profit over the remaining execution time;
 - 2: Let t denote the job (fragment) that triggers the interrupt;
 - 3: **if** $\rho(t) > u \cdot \rho(s)$ **then**
 - 4: Preempt s and insert it back to the priority queue H ;
 - 5: Execute t ;
 - 6: **else**
 - 7: Discard t ;
 - 8: **end if**
-

4.7.2 Profit-Dominant Strategy

In DD strategy, jobs are prioritized according to laxity first. And when some jobs with significantly higher values are about to expire, they are given a second chance to be considered for execution. A dual approach would be to prioritize jobs according to profit density first. If some jobs with less profit density are about to expire, they may also preempt jobs with higher profit density values. We call it a *Profit-Dominant* (PD) strategy. Similar to DD strategy, a PD strategy can also be implemented through a *NewJob* interrupt (or a *NewInput* interrupt if using *OptProfit* to evaluate job profit density) and a *LastDitch* interrupt (refer to Algorithm 8 and Algorithm 7). The key difference is that now in the handler *LastDitch*, the threshold u is less than 1 since the preempting job bears less profit density.

Algorithm 8 The *NewJob* Handler for PD Strategy

Handler_PD_NewJob:

- 1: Let J denote the set of jobs that triggers the interrupt;
 - 2: $\text{CompJob}(J)$; /* Compute the deadline laxity and profit density for each of the job in J */
 - 3: Remove jobs with negative laxity values from J ;
 - 4: Insert jobs in J into a priority queue H . H is sorted in non-ascending order of profit density. Ties are broken by choosing the job with less laxity;
 - 5: **if** the profit density of the first element in H is more than the current running job's profit density **then**
 - 6: Preempt the current job and run the first job in H ;
 - 7: **end if**
-

4.8 Intelligent Tuple Batching

Scheduling strategies bring considerable overhead to a stream scheduler. This makes tuple batching absolutely necessary. By grouping tuples together as a single scheduling unit, the overhead can be substantially reduced. Tuple batching has been implemented in the Aurora scheduler. But there is not much discussion on how batches are constructed in their work.

By default, when an operator is scheduled, naturally all the tuples that are pending in the input queue form a batch. This is, however, not necessarily a good way of constructing batches. We argue that tuple batching strategy needs to be carefully designed as our experimental study shows that the scheduling results to a certain extent depend on tuple batches. For example, if tuple batch size is too small, there will be too many reschedulings and context switch overhead. On the other hand, if batch size is too large, scheduling accuracy may drop because tuples of the same batch may have very different expiry dates. Given the dynamic nature of data input, to statically determine batching size is deemed inappropriate. Hence we propose a dynamic criterion to group tuples: Sequential tuples from the same input may form a single batch if 1) the timestamp difference between the head tuple and the tail tuple in the batch does not exceed $\alpha\mu$, where μ is the average laxity of the jobs currently in the system and α is a runtime coefficient and 2) the timestamp difference of any two consecutive tuples is no more than $\beta\tau$, where τ is the average inter-arrival time of the corresponding input stream and β is a runtime coefficient. Both μ and τ can be obtained by a statistical manager in a stream processing system. Criterion 1 essentially constrains the length of the batch. The reference metric

is the average laxity of the jobs currently in the system. The intuition here is job's laxity should be positively related to the length of the delay that input tuples can tolerate. Criterion 2 essentially determines the point that can mark the end of a batch.

4.9 Experimental Evaluation

4.9.1 Experimental Setup

We evaluate and compare six scheduling algorithms (*Basic*, *OptProfit*, *Basic*+DD, *OptProfit*+DD, *Basic*+PD, *OptProfit*+PD) discussed in this chapter using our DSMS prototype system. The *Basic* strategy is a naive greedy strategy discussed in Section 4.6.1. It works in a very similar way as the Aurora scheduler. The *OptProfit* (Section 4.6.2) is an improved greedy strategy. *Basic*+DD and *OptProfit*+DD are two Deadline-Dominant strategies (Section 4.7.1) integrated with *Basic* and *OptProfit*, respectively, for evaluating jobs' profit densities. Similarly, *Basic*+PD and *OptProfit*+PD are two Profit-Dominant strategies (Section 4.7.2) integrated with *Basic* and *OptProfit*, respectively. Besides, we also implemented a round-robin scheduler as a baseline. The DSMS prototype system mainly consists of three components: query engine, statistics manager and query scheduler. The query engine is able to process queries involving selection, projection, join and aggregation. The statistics manager monitors information such as the unit processing cost of each operator and selectivity as well as the current QoS of each registered query, and reports them to the scheduler, which then makes scheduling decisions based on the

information.

The queries used in the experiments are generated randomly. The number of queries ranges from 20 to 28, and the number of operators is between 32 and 48. The query tree is generated in a bottom up fashion. Given the number of queries, the same number of leaf operations are generated first, one for each query. Then, a few operators are assigned to be the parents of these leaf operators. The fan-out of each parent operator is generated according to the normal distribution with mean equal to the average fan-out of the query tree. The type of each operator is also assigned randomly as one of the followings: select, project, join and aggregate. Each query is also given a weight factor (an integer between 1 and 10).

We use both real and synthetic data for our experiments. The real data set is a trace named “LBL-PKT” [74], collected from the Internet Traffic Archive [2]. It contains an hour’s worth of all wide-area traffic between the Lawrence Berkeley Laboratory and the rest of the world. We split it into four parts as our input streams. The real data set is used for the experiments in Sections 4.9.2.1, 4.9.2.2 and 4.9.2.3. In Section 4.9.2.4, a synthetic data set is used because we need input streams with customizable properties. The synthetic data is produced according to the *b*-model [96], an easy and efficient way to simulate web and network traces with bursty and self-similar property [59]. For each input stream, we also assign a validity period which indicates when tuples get expired. The validity period essentially sets the maximum tolerable delay of producing a qualified output. The value is chosen randomly from 4s to 8s.

All the experiments were conducted on a 64-bit machine with an Intel Xeon 2.66GHz CPU and 8G RAM.

4.9.2 Performance Study

We use QoS score (computed using Equation 4.3 in Section 4.3.1) to evaluate the performance of the scheduling strategies. When all the output tuples are delivered in time, the score will be a perfect 100%. The round-robin scheduler is used as our benchmark strategy in some of the experiments.

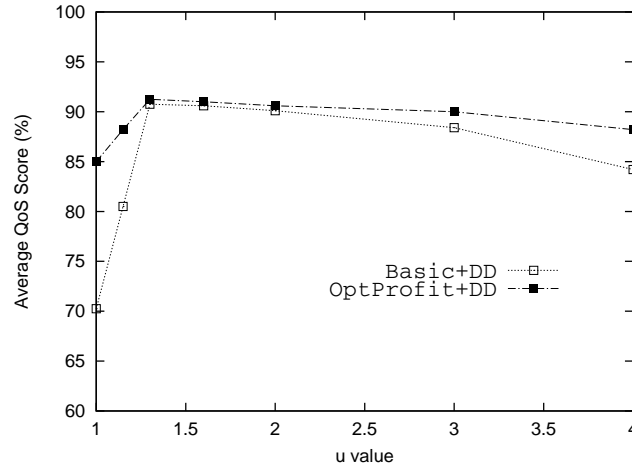


Figure 4.6: Coefficient u (DD)

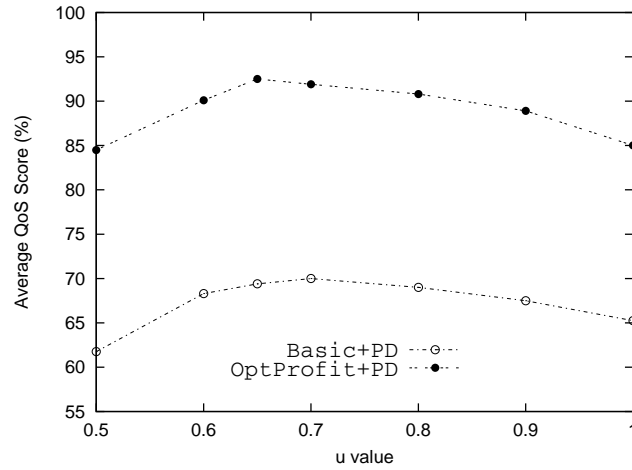


Figure 4.7: Coefficient u (PD)

4.9.2.1. Coefficient u

Before evaluating the performance of the various scheduling strategies, we first need to configure the coefficient u which is needed by all the deadline-aware strategies in the *LastDitch* interrupt (refer to Algorithm 7). Recall that u sets the threshold which determines how much a job's profit density is large enough to preempt the executing job. As mentioned before, the appropriate u value for DD strategy should be greater than 1 while for PD strategy the value should be less than 1. Figures 4.6 and 4.7 show the QoS score of the DD and PD strategies, respectively, as u changes. We can see that for both *Basic*+DD and *OptProfit*+DD, the QoS is the highest when u is around 1.3. While for PD strategies, the u value should be set to around 0.65. Because the threshold value is affected by several factors such as query weight distribution and system workload, the optimal choice of u differs from one query to another. Though it may be hard to determine the optimal u for a given scenario, it is nevertheless not difficult to obtain a quasi-optimal result. In this case, for DD any value between 1.2 and 3, and for PD any value between 0.6 and 0.9 should be considered good choices. We will set u to 1.3 for DD and to 0.65 for PD for the rest of our experiments.

4.9.2.2. Tuple Batching

In Section 4.8, we introduced our intelligent batching technique for minimizing scheduling overhead while keeping the scheduling accuracy at a satisfactory level. We verify the effectiveness of the technique by comparing it against three other strategies: “no batching”, “fixed length batching” and “fixed interval batching”. “No batching” means the scheduling is performed purely at tuple

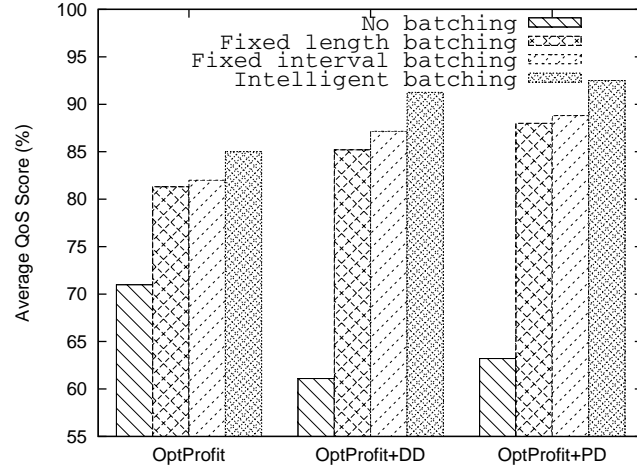


Figure 4.8: Tuple batching

level. In other words, each tuple triggers an interrupt. In “fixed length batching”, all batches have the same number of tuples. The batch length is set to be the average batch size when intelligent batching is used. “Fixed interval batching” is similar to “fixed length batching” except that now instead of fixing the batch length, we fix the batch interval. That is, the *NewJob* (or *NewInput*) interrupt is triggered for every fixed interval. Both “fixed length batching” and “fixed interval batching” can be seen as static batching strategies. The results reported in Figure 4.8 clearly tell us two things. First, tuple batching is essential. Although “No batching” achieves absolute tuple-level scheduling, the overhead is simply too high to be viable. Secondly, our proposed intelligent batching clearly outperforms the static batching techniques. Note that a consistent improvement of 5% or more in QoS score should be considered significant. In financial industry, for example, a 5% increase in QoS may directly amount to 5% more clients or profits. The run time coefficients α and β required by the intelligent strategy is set at 0.9 and 4, respectively, in the

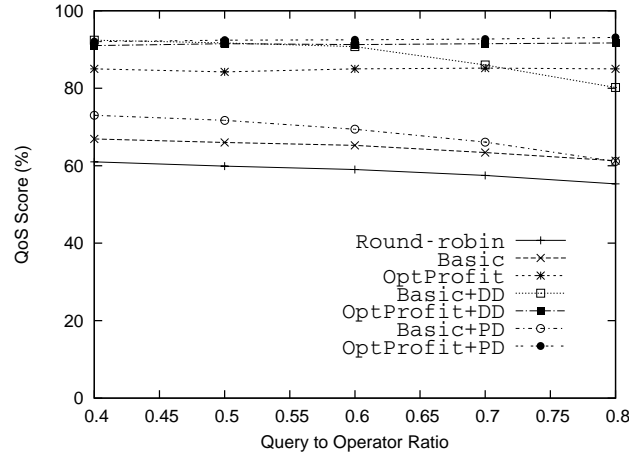


Figure 4.9: Operator sharing

experiment. Similar results can still be obtained if we vary these parameters by no more than 30%. The above observation is also applied to *Basic* and its related strategies (*Basic+DD* and *Basic+PD*).

4.9.2.3. Query Influence

Operator sharing For large scale multi-query data stream systems, operations are often shared among queries to minimize the total costs by avoiding redundant work. This results in tree structured query plan which greatly increases the scheduling complexity. In this experiment, we would like to see how each strategy performs in a multi-query environment with complicated operator sharings. We use the ratio between the number of queries and the number of operators to measure the degree of operator sharing. Figure 4.9 shows *OptProfit* and its related strategies consistently outperform other strategies. This is especially true when the degree of operator sharing is high. It indicates that the *OptProfit* strategy offers better scalability with respect to the complexity of a query graph.

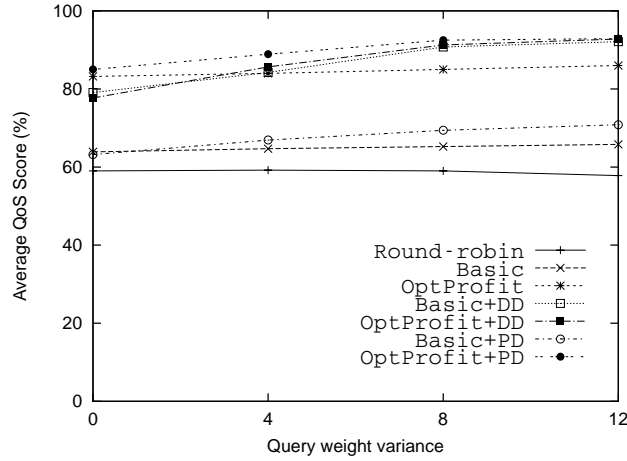


Figure 4.10: Query weight variance

Query weight Query weight can influence scheduling accuracy, too. We use variance to measure the degree of variation among jobs' weight. Figure 4.10 shows that DD strategies are particularly sensitive to the query weight distribution. When the job weight variance is low, they perform fairly poorly. However, as the variance increases, the QoS scores of DD strategies quickly pick up. This is interesting as it suggests that DD strategies are more useful when query weights are more heterogeneous.

4.9.2.4. Input Influence

Ideally, a good scheduling strategy should work well regardless of the input conditions. In this section, we evaluate the performance of the scheduling algorithms with different input workload and data characteristics.

Input load We first conduct the stress test by slowly increasing the input data rates and see how each scheduler responds to it. Figure 4.11 shows that the *OptProfit* schedulers generally perform better. Compared to other schedulers, they achieve graceful degradation as the system approaches overload. This is

4.9. EXPERIMENTAL EVALUATION

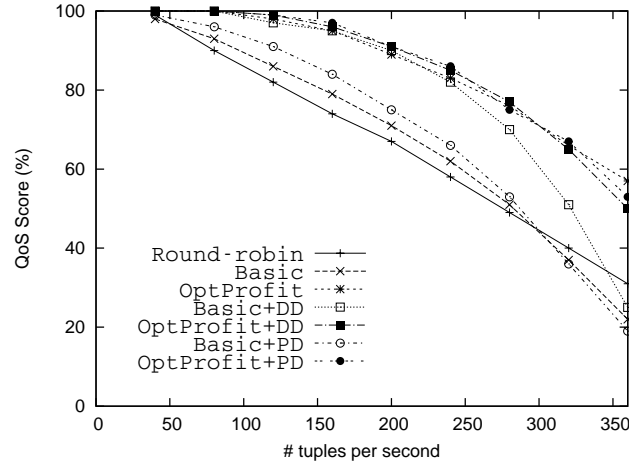


Figure 4.11: Response to input load

mainly attributed to the fact that *OptProfit* algorithms adopt *NewInput* interrupt handling which incurs relatively fewer number of reschedulings compared to *NewJob* interrupt handling used by *Basic* and its related strategies. Fewer reschedulings amount to less overhead, which can be crucial for systems with heavy workload.

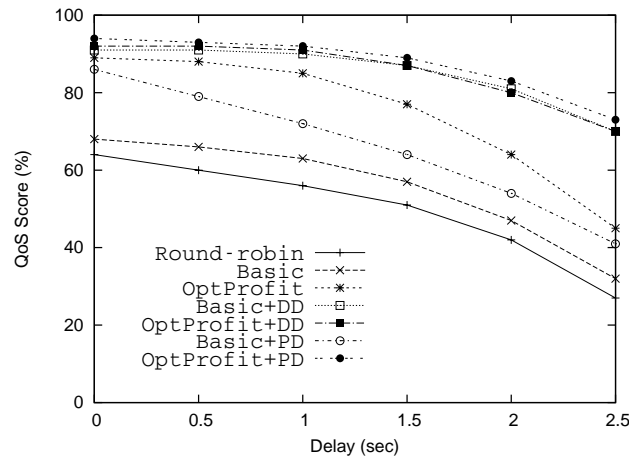


Figure 4.12: Response to tuple urgency

Tuple urgency Instead of increasing input data rates, we stress the

4.9. EXPERIMENTAL EVALUATION

system by introducing extra delay on input tuples. The delay can be seen as a simulation of input transmission latency from the data source to the query engine. An increase in delay results in a decrement in jobs' average laxity. We observe that deadline-aware strategies are advantageous in this scenario. When more jobs are about to expire, the *LastDitch* interrupt effectively saves many of them (particularly those with higher weight) from being late. This explains why these approaches do not degrade as much as the others when tuple urgency increases (refer to Figure 4.12).

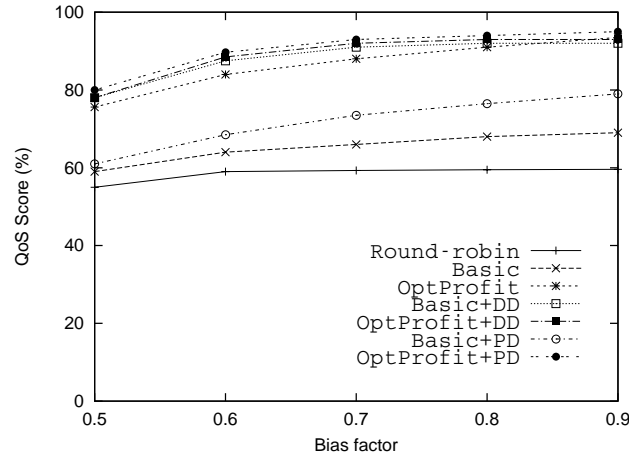


Figure 4.13: Response to bias

Data pattern It is known that network data, as well as other forms of data stream traffic, often exhibits burstiness and self-similarity. Therefore, it is important to study how the schedulers respond to such data pattern. We use the *b*-model [96] to simulate self-similar traces. The idea is quite straightforward. It first divides the total trace generation time into two equal intervals. The first interval will contain either b or $(1 - b)$ portion of all tuples, determined randomly. The parameter b is called the bias. The rest of the tuples go to the second interval. Then, each interval is further divided into two subintervals,

and each subinterval receives either b or $(1 - b)$ portion of the tuples in that interval. This process repeats until the interval is reduced to the unit time interval of the given resolution. The value of b ranges from 0.5 to 1. When $b = 0.5$, the trace generated will be evenly distributed over time. As b increases, the trace becomes burstier; in other words, the timestamps of the input tuples tend to be more clustered. It has been shown that the burstiness of real network traffic data can be reproduced using the b -model with b between 0.6 and 0.8. Figure 4.13 reports the QoS scores of the various strategies as b moves from 0.5 to 0.9. To our surprise, most of the strategies favor bursty input. Their QoS scores increase when b becomes larger. A second thought suggests two reasons for this. First, the average batch size increases as the input becomes burstier. Given the same number of total input tuples, an increased batch size means less frequent reschedulings, and consequently less overhead associated with each strategy. Secondly, a larger batch size means a scheduler can look further as it obtains more information about the future workload; this reduces the chance of scheduling a suboptimal plan or the probability of job preemption in the near future. Although all the algorithms improve as input becomes burstier, the *OptProfit*+PD strategy appears to be the best choice.

4.10 Strategies in Retrospect

From a real-time system perspective, a desirable scheduling strategy should offer *high scheduling accuracy* with *low computation overhead*. In this section we use this criterion to review the algorithms discussed in this chapter.

As online algorithms, their overall scheduling accuracy can be roughly as-

sessed by two factors: 1) the optimality of the schedule generated at each scheduling point and 2) the responsiveness with respect to job expiry. We have shown that, given a set of jobs with precedence constraints, the *OptProfit* algorithm generates an optimal plan which maximizes the total profits. That is why *OptProfit* and its related strategies (*OptProfit*+DD and *OptProfit*+PD) generally produce better QoS scores in most of our experiments. In contrast, the plans generated by *Basic* and its related strategies are suboptimal. In terms of the responsiveness to job expiry, pure greedy approaches (i.e. *Basic* and *OptProfit*) perform poorly because their scheduling decisions are based on the job profit density only. Deadline-aware approaches (including both DD and PD), on the other hand, tend to strike a good balance between profit density and job deadline. In particular, the experiments suggest that PD strategies are often the best option, especially when it is used in conjunction with the *OptProfit* algorithm for evaluating jobs' profit density.

The scheduling overhead can be quantified approximately as the production of two parameters: 1) the unit scheduling cost and 2) the frequency of rescheduling. The unit scheduling cost refers to the cost of running the scheduling algorithm which triggers at each scheduling point. It is a function of the algorithm's complexity. The frequency of rescheduling refers to the number of occasions where the scheduler needs to be re-invoked to update the job execution plan. The *Basic* strategy has a relatively low unit scheduling cost since it uses an approximation method to schedule jobs which runs in $O(n \times \log n)$. Comparatively, the more complicated *OptProfit* algorithm runs in $O(n^2)$. In terms of the frequency of rescheduling, the *OptProfit* strategy should be much lower because rescheduling is only required at *NewInput* interrupt (compared

to the rescheduling at each *NewJob* interrupt in the *Basic* strategy). Both deadline-aware strategies are expected to incur high frequency of rescheduling as well because they have one additional type of interrupt to handle, the *LastDitch* interrupt.

4.11 Summary

In this chapter we advocate tuple-based data stream scheduling from a real-time system perspective. By translating stream scheduling to job scheduling, we discovered optimization opportunities that would have not been found otherwise. Particularly, we found the existing greedy based approach only covers part of the solution space for our problem and hence proposed several new strategies. Experimental study shows that our new proposed algorithms are generally superior to the traditional approach though it may be difficult to find a clear winner strategy which outperforms the others in all situations. One possible future direction is to study the possibility of using a hybrid scheduling strategy which embraces various scheduling policies we studied. The hybrid strategy is expected to intelligently change its policy according to the query environment (query type, system load, input characteristic, etc.) to achieve the best possible result.

5

Scientific Sensor Data Management: A Case Study

One important force that drives the development of data stream research is the fast-growing Wireless Sensor Network (WSN) applications that have a profound impact on our lives. For example, now we are able to forecast some of the life-threatening natural hazards through real-time environmental monitoring powered by the state-of-the-art wireless sensor network technologies. In this chapter, we take a scientific sensor data application as a case study to explore data management issues in WSN. The chapter can be logically divided into two parts. The first part (Sections 5.1 to 5.9) is a self-contained description of an integrated data processing engine we developed specifically for scientific sensor data management. The second part (Section 5.10 and 5.11) discusses how the stream processing techniques introduced in the previous chapters can benefit such an application.

5.1 Background

Environmental monitoring data collected from wireless sensors typically need to be further processed before being utilized for scientific research. This is because raw sensor data are noisy and incomplete, and hence need to be cleaned. More importantly, there is a mismatch between what scientists desire from the data and what raw sensor data can offer.

Unlike traditional DBMS where users ask for information that can be directly looked up from the database tables, scientific queries are more analytical. The raw input data have to be interpreted with mathematical or geostatistical models provided by the users before they can be used for query computation. We refer to such a step as “data preparation”. Note that such data preparation is not a one time job. It is required to be adapted based on the requirements of the user queries. Also, scientists often try to interpret data with different models to see how the query output would react. The traditional Clean-Store-Query paradigm hence cannot be applied here. It is desirable that the data preparation step can be inserted during the query phase.

As there is a lack of a general framework to embrace all the necessary data processing, scientists often use diverse customized codes and various tools for different processing tasks. As such, the whole processing procedure is usually conducted in a number of separate steps. As we will see later, such an approach cannot exploit the opportunities of optimization across multiple steps and hence often leads to inefficient scientific query processing. Furthermore, the lack of a generic processing framework also prohibits the application of generic optimization techniques. Only ad-hoc optimizations written by cus-

tomized codes are possible. Finally, scientific data processing often involves visualization products, and progressive visualization is a desirable feature for many science applications. However, the multi-step processing approach limits the extent of progressive computation and visualization that can be exploited.

For these reasons, we propose an integrated and easy-to-use data processing system for environmental scientists to alleviate their burden of sensor data manipulation. Our work is inspired by the success of the relational DBMS technology which provides an integrated and efficient business data management platform by offering a data model and a generic query processing and optimization framework. Hence, in the system we built, called *HyperGrid*, we also include a data model, a query processing framework as well as several generic optimization techniques.

Our context for studying scientific data processing is the SensorScope project [19], which features a wireless sensor network that produces spatial and temporal measures for ecological and environmental monitoring. The system consists of multiple solar-powered sensing stations that measure key environmental data such as air temperature, humidity, solar radiation, and wind speed and direction. These sensing stations periodically sample their sensors and transmit the readings through wireless channels to the central server. Scientists can then retrieve the data through the central server in real time. Because these data are still in the rudimentary form, a series of transformations has to be performed before they are ready for scientific research.

The *HyperGrid* system is designed and tailored to such scientific applications. It offers an integrated environment for managing scientific sensor data. The logical abstraction provided by *HyperGrid* significantly saves users' ef-

forts from handling low level data operations such as array manipulations and coordinates transformations. Furthermore, we show that such an integrated framework offers abundant opportunities for query optimization. In fact, we have implemented several optimization techniques in *HyperGrid* and their effectiveness has been verified in our experimental study.

5.2 Related Work

One important characteristic of environmental monitoring data is that they can be uniquely identified by the spatial and temporal attributes corresponding to where and when they are measured. Conventionally, array [78] is chosen as the basic data model to store scientific measurement data with temporal and spatial properties being the array dimensions. Structural regularity and concise representation make array-based model suitable for managing data involving complex computations. The database community has proposed quite a few data models and languages to support array-based data management. AQL [63] is a calculus-based language for supporting low level array operations. Similarly, AML [69] also proposes a few operations for array manipulations. These operations, including those proposed in [21, 22] focus on aspects such as index patterns or sub-sampling of the array elements. Although these operations are important in image processing applications, they are not so useful for managing scientific environmental data. There are also grid-based models proposed [49, 73]. However, as can be seen later, they are also not applicable to the application discussed in this chapter. The feature of the work in [49] is an algebra of manipulating irregular grids, while [73] focuses on indexing technique

for grid data. Moreover, although most of the scientific data, particularly monitoring data collected from sensors, carries important spatio-temporal identity, to the best of our knowledge, none of the existing array-based techniques takes that into account during operations. In MauveDB [36], the author proposed a model-based view approach to manage measurement data. While our notion of perspective (which will be introduced later) shares the same flavor as the view in MauveDB, the two are in fact quite different. MauveDB focuses more on view maintenance issue to provide a consistent view to the user. Our approach, on the other hand, focuses primarily on query processing and using perspectives to implement and optimize scientific operations.

Scientific queries are often analytical. Hence, they typically involve data grouping or aggregation. Literatures on multi-dimensional OLAP, such as [4, 30, 65], have offered abundant techniques to speed up the performance for these queries. In general, these techniques are difficult to be applied in our context. The reason is OLAP optimization techniques usually rely on the fact that the attributes for grouping by are known or at least deterministic before the query is issued. However, in our case, scientists can freely organize the data in the continuous spatio-temporal domain to form a query. Moreover, external physical model may be introduced to interpret or reorganize the data. These make the optimizer difficult to perform any pre-computation to improve the query response time. Data used in aggregation can also be modeled as volume [28, 29, 97], where spatial and temporal dimensions are treated indeed as continuum. The related techniques may be useful as a supplementary approach to support arbitrary grid granularity for HyperGrid in our future work.

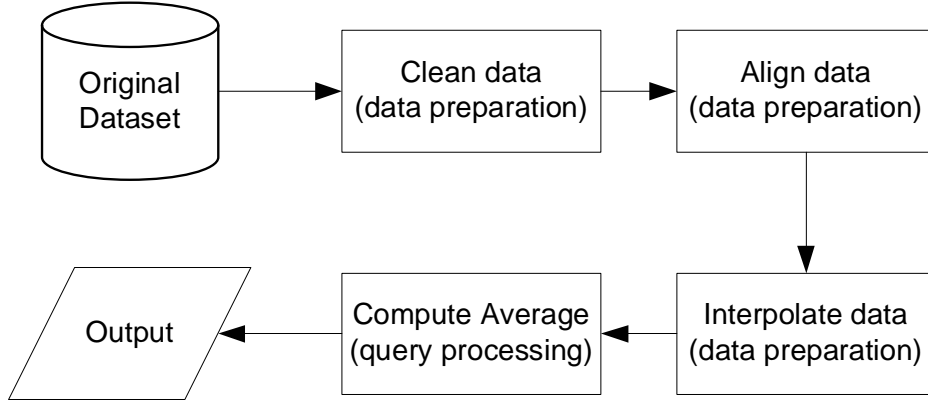


Figure 5.1: Work flow of Example 5.3.1

5.3 Motivating examples

Before embarking on the details of the system, we first give motivating examples which describe two representative queries issued by scientists and explain why processing these queries are poorly supported by the existing techniques. The first one illustrates a routine query which computes some statistical information about the dataset. The second example depicts a scenario where information is queried for the purpose of data exploration.

5.3.1 Scenario One

Consider the following query that a scientist typically issues:

Example 5.3.1 *Return the average ambient temperature over the period from 2007-10-01 00:00 to 2007-10-04 00:00 for the region $[45^{\circ}52'1''N, 45^{\circ}52'23''N]$ in latitude and $[7^{\circ}10'37''E, 7^{\circ}10'59''E]$ in longitude on a $1'' \times 1''$ grid.*

What the scientist has available is raw temperature readings collected from sensors. Before the data can be used to answer the query, they need to go

through several preprocessing steps. Firstly, the original dataset needs to be cleaned. Corrupted points are removed or replaced. Secondly, because sensor data are not available at all the locations specified in the query, the original data need to be interpolated over the geographical space. However, in order to do this, data values have to be first aligned on all dimensions except for those involved in the interpolation. In this example, data need to be aligned over the time dimension before being interpolated on the spatial dimensions (which means the set of data points involved in the same interpolation computation must either have the same temporal value or fall into the same interval). Methods to align the data include to randomly pick one representative data reading (i.e. sampling) for each aligned interval or to take an aggregation over the aligned interval. Once this is done, the spatial interpolation can be performed with the granularity $1'' \times 1''$ for each aligned time slice. Only at this stage is the curated dataset ready to answer the query, which is a simple average aggregation over time from 2007-10-01 00:00 to 2007-10-04 00:00 for the region between $[45^\circ 52' 1''\text{N}, 45^\circ 52' 23''\text{N}]$ and $[7^\circ 10' 37''\text{E}, 7^\circ 10' 59''\text{E}]$. A flowchart describing the steps of producing the output is given in Figure 5.1.

It is interesting to note that most of the efforts are actually spent on preparing the data rather than answering the query. This is indeed quite common in scientific data processing. In fact a complete query specification should also include details on how the data are prepared because scientists can have various ways to preprocess the data and their choice of procedure directly influences the query results. There is, however, no standard formula on how the data should be prepared towards a given type of query. Parameters such as the size of the interval in data alignment are determined by knowledge experts and are

subject to change. Unfortunately, relational DBMS cannot directly support many data preparation operations with configurable parameters. Although array database is able to achieve this, the separation between array elements and their spatio-temporal context makes the process tedious, especially when operations involve external models. Take interpolation as an example. Kriging as a popular geostatistical technique is often used for interpolation. It requires a variogram model to describe the degree of spatial dependence for a given region. Because the spatial coordinates of the data points in an array database cannot be directly obtained, such spatial dependence hence requires extra effort to derive by stitching the data points with the variogram model. Other operations involving external physical models also face the similar problem. In short, a pure array-based implementation is deemed inadequate for supporting such scientific queries.

5.3.2 Scenario Two

The second example demonstrates a scenario that occurs frequently during data exploration. In this case, scientists want to study the factors that impact the solar radiation for some region. They hypothesize that the rainfall rate may be the major influencing factor, which can be negatively correlated to the solar radiation value. They want to verify their hypothesis through the following query:

Example 5.3.2 *Compute the hourly average rainfall rate and hourly average solar radiation over the period τ at some point A . Display the results by plotting a diagram for each type of the measurement.*

To focus on the essence, we omit the details of the time and the location in this query. The query itself is not much different from the previous example except now the result needs to be displayed for visualization. In a data exploration scenario like this, scientists often do not need an accurate quantification as the answer. Instead, they prefer the results to be plotted so that they can see intuitively whether the outputs are as expected in general. A similar scenario also emerges when scientists want to study a new physical model. In that case, a bunch of similar queries, differed only by the values of the model parameters under investigation, are executed. Scientists want to gain a quick understanding of how these parameters influence the behavior of the model. To achieve this, the processing engine must facilitate results to be generated in a progressive way to cater for the interactive visual exploration. We will return to this topic when optimization techniques are discussed in Section 5.7.

5.4 Data Model

As mentioned, spatial and temporal attributes are important components of scientific data. First, they are used to identify each data reading. More importantly, they can be treated as the intrinsic properties of the associated measurements, through which scientists can reason about the meaning of the measured data as well as the physical phenomena implied by them. However, if a pure array model is used to represent scientific data, the spatio-temporal associations with the measurements are lost. This could severely hinder efficient scientific sensor data manipulation. To this end, we extend the logical array data model and propose a new *HyperGrid* data model. Scientists have been using grid

to model and organize their data. Experience shows that grids work well for typical applications, provided input can readily fit onto the grid points. Data abstracted from grids can be ordered, filtered or aggregated according to the given criteria without much difficulty. Our proposed *HyperGrid* model is based on a grid structure which can accommodate both rudimental sensor readings and high level data abstraction within the same framework.

Essentially, a HyperGrid can be seen as a collection of overlay grid structures built on top of a scientific dataset. Each overlay grid¹ is called a *perspective*². All *perspectives* in a HyperGrid are derived from a single *base perspective*, which is a special *perspective* that links raw data readings with a grid construct. The *base perspective*, or simply *base*, has a predetermined data structure which sets the coordinates and dimensions of the HyperGrid in a multidimensional space. The grid granularity of a *base* is fixed in line with the resolutions of the measuring devices for the corresponding spatial or temporal dimensions. This implies any generated spatio-temporal coordinates can be precisely captured in *base*, ensuring lossless mappings from sensor readings to the corresponding grid points.

On top of the *base*, a set of *perspectives* can be defined by the user or inferred by the system to reflect the user's views over the data. These *perspectives* typically have a coarser grid granularity than the *base*. There is no limit on the number of *perspectives* in a given HyperGrid. Users can create as many

¹Unfortunately, the term “grid” may also denote “grid computing” in computer science. This is not what we mean here. A “grid” in our context refers to a data structure (similar to “mesh”) for managing scientific dataset.

²A perspective in some ways is analogous to a view in traditional DBMS. However, we deliberately use a different term here to distinguish it from a database view because the role a perspective plays in HyperGrid is fundamentally different from the role a view plays in a DBMS.

perspectives as they want for their purposes.

The traditional grid structure only models objects in spatial domain. In *HyperGrid*, each *perspective* also includes time as an additional dimension in the grid space. From a data model point of view, temporal dimension is treated no differently from a spatial dimension. However, scientific queries over temporal space are more involved in query semantics especially for aggregations. We will discuss these issues later in detail.

Like the traditional grid, each *perspective* is composed of two parts: a grid topology and data values associated with it. The topology refers to the layout of a grid. Essentially it defines how data are grouped along each spatial and temporal dimension. The grid consists of cells that are regularly placed according to the topology definition. Each cell can be seen as an abstract object that represents certain spatial and temporal span. It is important to note each cell is identified by the spatial and temporal coordinate of its “lower-left” corner (imagine cell as a rectangular or an orthotope in a multi-dimensional space), rather than their relative position index as in the array-based model. This is a significant distinction between array and perspective. The coordinate not only uniquely identifies each cell within a perspective, but more importantly, it associates cells among different perspectives through their spatio-temporal context. As we shall see, this can play a very important role in scientific computations.

A *perspective* is only instantiated when the grid topology is bound with data. However, in what follows, we abuse the notation P_i to denote both a *perspective* and its grid topology when there is no ambiguity. Hence, the data value associated with a cell $e \in P_i$ can be represented by $P_i[e]$. There are various types of data values depending on different dimension aggregations

they entail. As an example, data values for hourly average temperatures at a given point in a geographical space is a 1-D aggregation because it only takes aggregation on one dimension (i.e. the time dimension). Intuitively, data values associated with an n -dimensional *perspective* can be from 0-D aggregation up to n -D aggregation. The only exception is *base*, whose associated data must be 0-D as it only stores data samples directly from the measuring devices. Our current implementation allows each *perspective* to associate one type of data value only. This makes transformations among *perspectives* neat and easy to manipulate.

5.5 Operations

Following the approach of DBMS, we also try to propose a generic data processing framework so that scientists could easily compose their routine data processing tasks and, as will be seen later, some generic optimization techniques could be applied to boost the processing performance. However, unlike the operations in traditional DBMS, processing scientific data requires high degree of customizability. This is actually one major reason why DBMS is not prevalent in scientific applications. Therefore, we endeavor to design generic but customizable operators, with which scientists could fill in their customized functions to form the specific operator they need. By doing this, we can keep the benefits of having a generic processing framework while providing the necessary customizability.

HyperGrid adopts a transformation based framework; scientific data processing is modeled as transformations among different perspectives. Hence, the

essence of HyperGrid is a sound and flexible perspective construction so that common data operations can be natively supported. In this section, we first describe the details of building a perspective, followed by an example to illustrate how common operations are supported by such a construct.

5.5.1 Perspective Construction

The construction of a new perspective (called target perspective) P_t typically requires one³ reference perspective (called source perspective) P_s from which the new perspective is derived. To be clear, we will use subscripts t and s to distinguish *target* and *source* perspectives respectively. Cells in *target* and *source* perspectives are correspondingly referred to as *target cells* and *source cells* in the rest of the chapter. Occasionally, the target and source perspectives are also called child and parent perspectives, respectively, when the context deems appropriate.

At the very outset, the *base* is used as the reference to create the first target perspective. In addition to P_s , constructing a P_t may optionally require three pieces of information: a topological definition T_t , a data function D_t and an input selection function I_t (i.e. $P_t = \langle P_s, T_t, D_t, I_t \rangle$).

As mentioned before, a topological definition T_t gives the internal layout of a grid. It determines the size and dimension of the cells within a perspective. Depending on the type of the associated data values, a grid layout can have different meanings. For example, when the associated data is a 0-D aggregation, the layout simply sets the grid granularity. On the other hand, when the

³The only exception is perspective that implements *Merge* operation, in which case multiple source perspectives are required.

associated data is a k -D aggregation ($k > 0$), the layout also serves as part of the query semantic that instructs how data are grouped and aggregated. Notably, a cell in a perspective inherits the characteristics of a traditional grid cell which captures the structural regularity. However, the former embodies a broader definition than the latter. A cell in a perspective is generalized as a logical computation unit, which may not be visualized as a single block or orthotope in a multidimensional space. For example, a cell can refer to a set of unconnected blocks or orthotopes that collectively form a logical unit. Also, neighboring cells need not be disjoint or adjacent as in traditional grid. They are allowed to overlap or contain space between them (as in Example 5.9.1). We will see how this generalized notion of cell benefits query construction, especially for aggregation queries, later in Section 5.5.3.

Data function D_t is another important component for perspective construction. It implements a scientific operation by dictating how data are transformed from P_s to P_t . The input of D_t are values associated with a set of source cells. The output of D_t is the computed result for some target cell e_t . Various forms of data functions for popular scientific operations are also discussed in detail in Section 5.5.3.

The construction of a new perspective involves both topological transformation and data transformation. These are two closely related processes. Topology conversion from T_s to T_t is implicitly performed through the output to input mapping of the data function D_t . Notice the output of D_t corresponds to the value of a cell confining to the topology T_t . However, the input of D_t is from cells confining to the topology T_s . Although it is not always the case, for some operations an explicit user-defined input selection function I_t is needed to in-

struct how cells under T_s should be selected to compute a target cell under the topology T_t . The function I_t takes one target cell $e_t \in P_t$ as input and returns as output the set of source cells $\{e_s | e_s \in P_s\}$ that will contribute to computing the value for the cell e_t .

As a final note, when a query is formulated as a series of perspective transformations, the last perspective in the series, called *surface perspective* (or simply *surface*), defines the final query results. In addition to the parameters above, a *surface* has one more optional parameter called *clipping window*, which defines a scope in the spatial and temporal domains where only data points within the defined window are returned.

5.5.2 Relationship Between Perspectives

Input selection function ensures data computation is carried out on the correct data set. However, such function is often not necessary for constructing a new perspective as long as the defined data transformation is *Location Consistent* as defined below:

Definition 5.5.1 *Let $V(e)$ denote the scope of a cell e defined in the spatio-temporal domain. And let I_t be the input selection function for some data function D_t . The corresponding data transformation is said to be Location Consistent (LC) if the following Location Equivalent condition holds:*

$$V(e_t) = \bigcup_{e_s \in I_t(e_t)} V(e_s), \quad (5.1)$$

All other transformations that violate the Location Equivalent condition are

Target Perspective Parameter	Default value or rule
Source Perspective P_s	the <i>base</i>
Topology Definition T_t	T_s
Data Function D_t	n.a. (compulsory parameter)
Input Selection Function I_t	LC rule

Table 5.1: Default settings for perspective parameters

Operator	Perspective #	Topology Def.	Data Func.	Input Selection Func.
<i>Convert</i>	One	Default	User defined	Default (LC)
<i>Merge</i>	Multiple	Default	User defined	Default (LC)
<i>Interpolate</i>	One	New definition	User defined	User defined (LA)
<i>Aggregate</i>	One	New definition	User defined	Default (LC)

Table 5.2: Characteristics of perspectives for different operators

categorized as Location Across (LA) transformations.

The input selection function can be omitted for LC transformation because a target cell and its contributing source cells can be automatically paired through their Space-Time Identity. Hence, user-defined input selection is only required for perspective computed from LA transformations. Fortunately, as we shall see later, most of the operations belong to LC transformations. Hence, by exploring the important “location equivalent” relationship among perspectives, operations can be defined in a more concise way. Moreover, the query executor can also take advantage of the LC property to optimize the query execution, as will be discussed in Section 5.7.2.

5.5.3 Operators

HyperGrid provides users great freedom to create their own data operations through customized perspectives. We have described in the previous section

that the definition of a perspective P_t is a quadruple $\langle P_s, T_t, D_t, I_t \rangle$. As examples, we show in this section how popular operations (convert, merge, interpolate and aggregate) in scientific sensor data processing can be readily supported by this construct. Although four parameters need to be supplied for the standard definition, in practice some of the parameters (such as I_t as described in the previous section) can be omitted by taking their default actions. Table 5.1 lists the default settings when the corresponding parameter is not specified. Table 5.2 summarizes the characteristics of the perspectives implementing these popular operations.

5.5.3.1. Convert

The *convert* operation converts data points in P_s to other values in P_t . The operation can be used in different ways for different purposes. One simple usage is to scale up or scale down values in the grid dataset by introducing a scaling factor in the data transformation rule. As another example, in data preparation phase, *Convert* can serve as a filter to clean corrupted sensor readings. This is achieved by converting erroneous data in P_s to “NULL” or some default values for the corresponding grid cell in P_t . A perspective that implements *convert* duplicates the topology of the source perspective (i.e. the default setting) since *convert* does not involve any structural change of the grid. Hence, other than P_s , the data function D_t is the only parameter to be specified, which can be formulated as follows:

Definition 5.5.2 *Given $T_s = T_t$, let C denote the conversion function. The*

transformation rule D_t is:

$$D_t(e) = C(P_s[e]), \forall e \in P_t \quad (5.2)$$

In the above definition, both P_s and P_t refer to the topologies instead of the entire perspectives. Because source and target perspectives share the same topology ($T_s = T_t$), for each grid cell e in the target perspective grid, we can find a corresponding data value associated with that cell in the source perspective.

5.5.3.2. Merge

Merge is the only operator which takes multiple perspectives as input. It is often used for producing a model that integrates multiple types of measurements, each represented by one source perspective. The operator enforces all source perspectives to have the identical topology and produces one target perspective with the same topology. Similar to *Convert*, the data transformation rule D_t is the only parameter to be customized, which can be defined as follows:

Definition 5.5.3 *Given N is the number of source perspectives ($N > 1$), and $T_t = T_{s_i}, \forall i \in N$. Let $d(e)$ denote the set of data values from $\{P_{s_i}[e] \mid i \in N\}$. The transformation function for Merge is:*

$$D_t(e) = \Gamma(d(e)), \forall e \in P_t \quad (5.3)$$

where Γ is a user defined function that merges the corresponding cells from each of the source perspectives.

5.5.3.3. Interpolate

In managing scientific data, especially environmental data, *interpolation* is such a popular yet expensive operation that deserves particular attention. As input are measurement readings, which are samples taken from continuously running physical processes (such as solar radiation and wind speed), without temporal interpolation it is very difficult to answer queries that ask for data at some point in time when no measurements were taken. Analogously, meteorologic phenomena monitored by WSN usually come with the “coverage-holes” problem owing to the sparsity of the network or nodes failure. In the SensorScope project, to set up a sufficient number of sensing stations in order to provide exhaustive coverage over a monitored region is infeasible due to prohibitive deployment costs. Hence, scientists also resort to spatial interpolation to generate a comprehensive data map for research and analysis.

Interpolation is a typical example of *LA* transformation. A perspective that implements *interpolation* defines its own grid layout T_t and data transformation rule D_t . T_t generates a set of new grid cells whose associated data values are to be interpolated. Computation for the interpolated points are defined by D_t , which comprises two steps. In the first step, a customized input selection function I_t is used to select candidate grid cells from P_s that will contribute to the computation for the grid cell in P_t . This is followed by applying a computation function to data values associated with the candidate cells to produce the interpolated result for the target cell in P_t .

Definition 5.5.4 *Let I_t denote the input selection function for interpolation and C denote the corresponding computation function. The transformation*

rule for Interpolation is:

$$D_t(e_t) = C(e_t, \Phi, \{P_s[e_s] \mid e_s \in I_t(e_t)\}), \forall e_t \in P_t \quad (5.4)$$

where Φ is a statistical model based on which the interpolated value is calculated.

5.5.3.4. Aggregate

Scientific data processing involves extensive aggregation operations for two reasons. Firstly, aggregation is used to compress sheer volume of data generated by the measuring devices to a manageable level. Secondly, scientific observations or assertions are typically supported by statistically significant data computed by certain aggregation functions rather than individual data readings.

Here we focus on aggregations with “group-by” clause on temporal or spatial attributes only since a predominant number of queries belong to this type. The HyperGrid model natively supports spatio-temporal data aggregation because n -D data in a perspective essentially represents the n -D volume of the corresponding spatio-temporal span defined by its associated grid cell. This implies that for an aggregation perspective, the target topology T_t constitutes an important part of the aggregation semantic. Notably, each grid cell is an abstracted spatio-temporal notion, which may not be necessarily visualized as a single block or orthotope as in the traditional grid. This generalizes the concept of grid cell and gives user great flexibility to construct the “group-by” criteria. For example, user may want to know the breakdown by each hour the average temperature for a given region for the past 30 days, e.g., the average temperature of the past 30 days between 00 : 00 and 00 : 59, between 01 : 00 and 01 : 59,

etc. Such queries are difficult to model by traditional grid constructs since each grid cell in the result set refers to 30 segregated spatio-temporal blocks which are evenly spaced by 24 hours in the time domain. A HyperGrid model allows multiple physically segregated blocks to form a single logical cell because in HyperGrid each cell is characterized by its spatial and temporal features, not just by a single cell boundary specification.

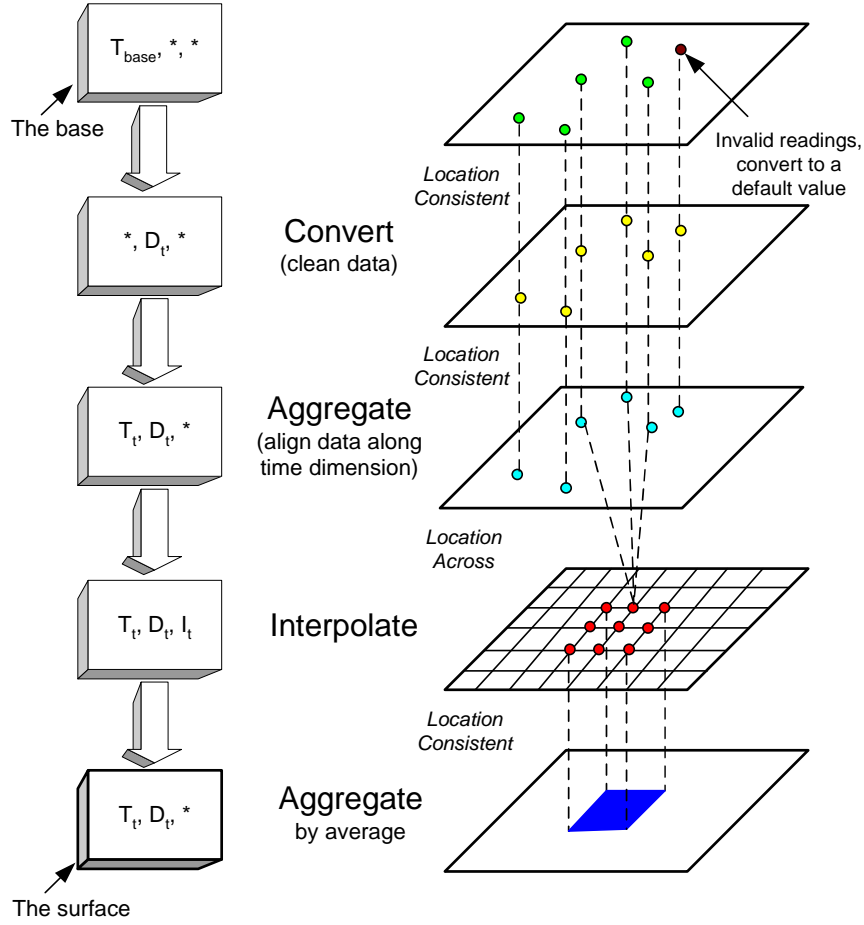
Like interpolation, the data transformation for an aggregation perspective is also a two-step approach. However, for the input selection step, user defined function is no longer required since aggregation belongs to LC data transformation. For the computation step, it simply applies an aggregation algebra (SUM, AVG etc.) to the candidate cells that transforms the k -D data in P_s to $(k + m)$ -D data in the corresponding cell in T_t , where m is the number of dimensions whose associated values are aggregated.

Definition 5.5.5 *Let $I_{LE}(e_t)$ denote the function that returns the set of source cells that collectively define the location equivalent scope as that of e_t in the spatio-temporal domain. And let C be the function that implements the aggregation algebra. Then the data function D_t can be formulated as:*

$$D_t(e_t) = C(\{P_s[e_s] \mid e_s \in I_{LE}(e_t)\}), \forall e_t \in P_t \quad (5.5)$$

5.5.3.5. Other operations

The perspective construction is a generalized notion which captures the transformation-based operations over grid data. In fact, the model is flexible enough to express more sophisticated operations other than the standard operations described above. Essentially, any grid-based operations that can be



Q1

Figure 5.2: Illustration of the query execution in Example 5.3.1

characterized as topological change, data change, or both are supported by the construct.

5.5.4 Illustrative Example

With reference to our previous motivating example 5.3.1, given the above definitions, we can organize the required operations into a query tree, which consists of a series of data transformations from the *base* all the way to the *surface* (i.e.,

the output) as shown on the left of Figure 5.2. Each box in the graph represents a perspective that implements one scientific operation. Arrows pointing from source perspective to target perspective represent the data flow. Inside each box, there are three parameters separated by comma. They represent, from left to right, topology definition, data function and input selection function. If a default is taken, the parameter is replaced by a “*” in the corresponding position. A graphical illustration of the query execution is shown on the right of Figure 5.2.

5.6 Query Execution Strategies

There are two basic execution strategies: “top-down” and “bottom-up”. The “top-down” strategy initiates the computation at the base perspective. The executor follows the data flow and materializes each of the perspectives one by one along the query tree. As an example, for the query plan in Figure 5.2, the executor first materializes the child perspective of the *base* (i.e. the *Convert* perspective), and then uses the obtained results as the source to compute the next level perspective and so on, until the *surface* is reached. However, a big drawback of this strategy is that the “top-down” computation leads to blocking execution; no output will be produced until the final surface perspective starts being materialized.

On the contrary, in “bottom-up” strategy, the computation begins at the *surface* and carries out upward in a pipelined manner: For each target cell in the *surface*, the contributing source cells in its parent perspective need to be computed first. Then, for each of these source cells, it in turn has to call

the cells in its parent perspective to get itself computed. This continues until the *base* is reached with the actual values getting returned. The whole process essentially resembles the iterator model [76] in traditional DBMS. Different from “top-down” approach, “bottom-up” strategy does not require any intermediate perspective to be materialized and it produces results in a progressive way (like online query processing). This is desirable because it allows scientists to terminate the processing prematurely if they are dissatisfied with the partially produced answer.

However, “bottom-up” strategy is not a very efficient approach (which will be explained in Section 5.7.2). Therefore, what *HyperGrid* actually adopts is a hybrid strategy which combines the “bottom-up” with the “top-down”. We call it *hybrid-k*, which means for a query plan with N perspectives, the top k perspectives in the query tree are computed first in the “top-down” manner while the lower $(N - k)$ perspectives are then computed using the “bottom-up” method. In fact, the pure “top-down” and “bottom-up” approach can be seen as the special cases of the *hybrid-k* strategy, where the “top-down” corresponds to *hybrid-N* and the “bottom-up” corresponds to *hybrid-0*, respectively. In Section 5.7.2, we will explain in detail why *hybrid-k* strategy is superior and how to determine the optimal k value for a given query plan.

5.7 Optimization Techniques

As scientific queries usually take as input an enormous amount of data and process them with expensive user-defined functions, how efficiently these queries can be executed becomes a critical issue. In this section we explore oppor-

tunities to optimize scientific query executions under the generic *HyperGrid* model.

5.7.1 Preprocessing and Query Rewrite

By preprocessing the original query plan, a series of perspective transformations can be rewritten in a succinct fashion. The goal of query rewrite is to produce a more economic plan that leads to reduced runtime costs without compromising the output quality. One effective approach is to coalesce adjacent perspectives in a query plan. The benefit of perspective coalescence is evident. Firstly, with fewer perspectives, the number of function invocations is reduced. Because the number of function invocations for each perspective computation is proportional to the number of cells in that perspective, minimizing the total number of perspectives leads to considerable savings in terms of function call overheads. Secondly, the amount of buffered intermediate results is also reduced with fewer number of perspectives. Scientific computations may generate intermediate data that are too huge to be buffered in the memory. Hence, the reduction of intermediate results may directly amount to the reduction in disk I/O.

Of course, it is not always possible to coalesce any pair of adjacent perspectives in a query plan. At least one of the perspectives has to be *coalesce-amenable* in order to ensure query results are not compromised therefrom. A perspective is said to be *coalesce-amenable* if it uses default topology definition and *LC* transformation. For example, any perspective that implements the *convert* is *coalesce-amenable*. A *coalesce-amenable* perspective is free to choose to combine either with its parent perspective or with its child per-

spective. The coalescence process involves two steps: 1) map the topology of the parent perspective to the topology of its child; 2) merge data transformation functions through function composition. For example, given a *coalesce-amenable* perspective $P_k = \langle P_j, "default", D_k, "default" \rangle$ and its parent perspective $P_j = \langle P_i, T_j, D_j, I_j \rangle$, they can be combined to form a new perspective $P_m = \langle P_i, T_j, D_m, I_j \rangle$, where $D_m = D_j \circ D_k$ (\circ denotes function composition). Similarly, if $P_l = \langle P_k, T_l, D_l, I_l \rangle$ is the child perspective of P_k , it is also possible to coalesce P_k with P_l to produce $P_n = \langle P_j, T_l, D_n, I_l \rangle$, where $D_n = D_k \circ D_l$. If the resultant perspective is still *coalesce-amenable*, it can continue to coalesce with its adjacent perspective. A query rewriter scans through a query plan and performs perspective coalescence until there is no more *coalesce-amenable* perspective existing in the plan or the plan is left with only one perspective.

For the query in Example 5.3.1, the query rewriter will coalesce the *convert* and *align* operators after the preprocessing.

5.7.2 Optimizing Query Execution

Section 5.6 has introduced two basic query execution strategies. The “bottom-up” approach is generally preferred over the “top-down” approach because the former allows query results to be produced in a progressive way. However, in practice, we find the “bottom-up” strategy may not be very efficient for two reasons. Firstly, it may lead to significant redundant computations. Secondly, a “bottom-up” execution may involve some “dull” computations which are useless to the query result. We explain these two issues and propose optimization techniques to tackle them in the following two subsections.

5.7.2.1. Iterator with buffering

The first problem with the “bottom-up” strategy is redundant computations. When a target cell in the child perspective requests the value for some cell in the parent perspective, the system would not know whether the same value has been computed before because nothing is saved or materialized in a “bottom-up” execution. Every data request will be computed from scratch following the iterator model. However, we show in this section that deliberate buffering strategy and intelligent choice of order in producing the cells of the surface perspective can effectively minimize such redundant computations.

Before delving into the details of the optimization techniques, let us first look at a strategy alternative to the basic iterator model. We attach a buffer for each intermediate perspective in the query tree. During the iteration, whenever a *NextCell()* function (analogous to the *Next()* function in an iterator) is returned, the results are stored in the attached buffer of the corresponding perspective. If, at a later time, the value of the same cell is requested again, the system can obtain the result directly from the buffer without recursively invoking the next level *NextCell()* function for the second time. Obviously, the buffering strategy avoids expensive redundant computations and hence reduces the query latency provided there are sufficient memory space to hold the intermediate results. Therefore, the crux of this approach is an efficient buffer strategy with low memory overheads and high hit ratio.

When a query only involves LC transformations, such buffer strategy is relatively easy to design, thanks to the topological regularity and cell’s spatio-temporal identity that effectively correlate the perspectives in a query tree⁴.

⁴It is worth noting that this buffer strategy is only meaningful when cells in the output

The spatio-temporal identity allows the system to identify cells in ancestor perspectives that contribute to the target cells to be computed. The topological regularity makes it possible to produce ordered output with respect to any spatial or temporal dimensions.

To ease exposition, we define what we call *Candidates Window (CW)* here. A *CW* defines a dynamic subspace of a perspective where cells subsumed by this space may potentially contribute to the future output. Notably, if all transformations are location consistent (LC), then perspectives along the query path will all share the same *CW*. In the beginning when the computation has not started, the *CW* is essentially the *clipping window* (refer to Section 5.5.1) defined in the *surface* and buffers attached to each intermediate perspective are empty. The buffers begin to be filled with intermediate results when the computation starts. For *CW*, it starts to shrink as more output cells have been generated. Eventually, the size of *CW* reduces to zero when computation completes. Because buffers attached to each perspective only need to cache results for cells contained in the current *CW*, the buffer manager regularly expires cells in the buffer whose location (identified by its spatio-temporal coordinate) has fallen out of the latest *CW*. This strikes a dynamic balance such that the buffer size remains stable. Obviously, the space efficiency of the above buffering scheme depends on how fast *CW* shrinks with respect to the growing intermediate results during runtime, which in turn is determined by the order of the output sequence. For example, if output cells are generated in time ascending

perspective are overlapping. Otherwise, no buffer is needed because intermediate results will not be shared among output cells that are disjoint. A query executor can easily determine whether cells in the output perspective overlap and decide the necessity of enabling the buffer strategy.

order, then *CW* will shrink steadily along time dimension from the lower end of the *clipping window* to the higher end during the query execution. Noticeably, buffer management using *CW* ensures optimized hit ratio because data discarded by the buffer is guaranteed not to be requested again by the subsequent computations. Also fine-grained buffer control is possible to improve space efficiency by taking multiple space-time dimensions as the sorting keys. In that case, the choice of dimension as the primary sorting key has the largest impact the total buffer size needed for the query execution.

When a query plan includes perspectives with LA transformations, however, optimal hit ratio can hardly be guaranteed for buffers corresponding to perspectives ascendant to the LA transformation perspective. That is because an LA transformation runs user-defined input selection function, which can choose any cells from its source perspective. This renders *CW*-based buffer strategy useless because the location equivalent property no longer holds. Nevertheless, we observe that most user-defined input selection criteria are not completely arbitrary. In fact, almost all of them exhibit certain locality property. This inspires us to use a *lookahead* heuristic to replace the *CW*-based buffer strategy for LA transformations. The idea is to run the input selection function in advance of the actual data computation for the target cells. By looking ahead the set of source cells that will be used to compute the next few target cells, the buffer manager can make intelligent decisions by only caching the results for the top k most referenced source cells (provided their results have already been computed previously). Owing to locality property, target cells in the vicinity are likely to share a big portion of source cells. This makes the *lookahead* approach practically effective in many cases.

5.7.2.2. The *hybrid-k* strategy

We also find the “bottom-up” strategy could sometimes involve “dull” computations which are useless to the query result. The reason for this has to do with the underlying structure for data storage. When a defined perspective contains a lot of holes (i.e. when the valid data point density of a perspective is low), for space efficiency the system will choose sparse array to represent that perspective, instead of an ordinary array. Note that an important characteristic of a typical HyperGrid query is that the density of valid data points of perspectives along a query path is often in non-decreasing order from the *base* to the *surface*. (Particularly, if a query involves interpolation, all descendent perspectives will have data point density of 100% since there will be no holes in the perspective after interpolating the space.) Hence, a typical scenario is that the system switches from sparse array implementation to ordinary array implementation for some perspective along the query path and continues to use ordinary array up to the *surface*.

Now consider the “bottom-up” strategy which iterates the computation from the *surface* to some perspective with very low data point density. It is very likely that the requested cell is a hole, which does not associate a valid data. However, under the “bottom-up” strategy the system would not know this, and the iteration therefore continues until the *base* is reached. As a result, some NULL values are returned and resources are wasted on computing something with NULL as input. In comparison, the “top-down” approach does not have this problem. This is because in “top-down” execution, perspectives are materialized **as a whole** one by one from the *base* downward. Computing a new perspective from a materialized sparse array only involves computa-

tions on those valid data points. One concern here, however, is that by using “top-down”, it violates our initial requirement of generating the results in a progressive way since “top-down” execution is a blocking process. Therefore, the *hybrid-k* strategy comes into the picture. Because perspectives near the top of a query tree can have very low data point density (typically less than 0.05), materializing them may not sacrifice much in terms of query responsiveness. The objective here is to strike a balance between “top-down” and “bottom-up” strategy (i.e. to find an optimal k value) so that the query’s average response time is minimized (the metric we use to measure user’s satisfaction). We first formally define the average response time for a given query as follows:

Definition 5.7.1 *Let $R(e)$ denote the latency from the time when computation for a query’s surface perspective starts to the time when one of its output cell e is produced, the average response time for that query is defined as:*

$$\frac{\sum R(e)}{n_{sf}}, \forall e \in P_{sf} \quad (5.6)$$

where P_{sf} denotes the surface perspective and n_{sf} is the total number of output cells in P_{sf}

In the “bottom-up” strategy, we have the following recurrence relationship:

$$\begin{cases} R(e_1) = r(e_1) \\ R(e_m) = R(e_{m-1}) + r(e_m) \end{cases} \quad \forall e_m \in P_{sf}, m > 1 \quad (5.7)$$

where $r(e_m)$ is the latency from the request to compute the cell e_m is generated at the *surface* until the result is returned. Intuitively, $r(e_m) = \sum_{i=1}^N r_i(e_m)$

where $r_i(e_m)$ is the time taken to compute e_m at perspective i and N is the total number of perspectives. Assume the buffering strategy described in the previous section is enabled. Also for simplicity, assume $r_i(e_m)$ to be equal for all $e_m \in P_{sf}$ (say, it is μ_i). Then the average response time using the “bottom-up” strategy can be estimated as:

$$\frac{(n_{sf} + 1) \sum_{i=1}^N \mu_i}{2}, \text{ N is the total number of perspectives} \quad (5.8)$$

On the other hand, if we choose to compute the first top k perspectives from the *base* in a “top-down” manner and the rest $N - k$ perspectives still using the “bottom-up” approach, we obtain the following average response time:

$$\sum_{j=1}^k \rho_j n_j \mu_j + \frac{(n_{sf} + 1) \sum_{i=k+1}^N \mu_i}{2} \quad (5.9)$$

Given $\rho_u \leq \rho_v, 1 \leq u < v \leq N$

In the above equation, ρ_j is the data point density of perspective j . It is a value between 0 and 1. n_j is the total number of cells in perspective j . As can be seen, whether a perspective should be computed “top-down” or “bottom-up” really depends on the value ρ . That is the density of the array for the corresponding perspective. Using equation 5.9, the optimizer can determine the optimal k value so that the average response time is minimized.

5.7.3 Optimization for Visualization

The previous sections introduce several optimization techniques to reduce the query execution costs or average response time. However, these are not the

only goals for an optimizer. Scientific research often involves tasks such as to discover unusual trend or pattern from a dataset or to collect evidence for supporting new hypotheses, etc. For these purposes, scientists need to filter useful information from numerous test cases by running a large number of exploratory queries or "what-if" queries. A pure cost-based optimizer is deemed inappropriate for such scenario especially under interactive mode where user is sitting in front of the monitor waiting for the results to be visualized. This is because in cost-based optimization, output cells have to be produced in an order by space or time dimension. This leads to results generated in a raster manner. A big disadvantage of this is user will not be able to get a rough idea of how the results look like until the majority of the cells have been computed. Probably the following comment well describes what is actually desired for a visualization output: "Overview first, zoom and filter, then details on demand" [83]. In our case, an overview means to provide the insight instead of the the accurate answer for each output cell. This implies the computation should prioritize "interesting" regions in the surface perspective that would help reveal global trend or unusual patterns etc. The definition of "interesting" here is context-dependent. But often, it refers to portions in the result set where data values have greater variations. An ideal executor should focus on these portions and progressively refine the answers if the user continues to be interested.

While a plethora of optimization techniques have been proposed for scientific visualization (see [84] for an excellent overview on the state-of-the-art techniques), we choose to propose a simple but effective algorithm for our application. The purpose is to show *HyperGrid* can facilitate efficient scientific visualization. More sophisticated visualization techniques may be included in

Algorithm 9 A directed random walk algorithm for visualization

Notations:

P_i : vector representation of the location of cell i

v_i : value of cell i

s : default stride for one step of walk

```

1: Randomly select a cell  $P_1$  with  $s$  distance away from the starting cell  $P_0$ .
2:  $i := 1$ 
3: loop
4:   if  $IsComputed(P_i) == false$  then
5:      $v_i := ComputeCell(P_i)$ 
6:   else
7:      $v_i := OutputBuffer(P_i)$  /* directly retrieve the results from output
      buffer without recomputing */
8:   end if
9:   if  $f(v_i, v_{i-1}) > threshold$  then
10:     $U := Normalize(P_{i-1} - P_i)$  /* unit vector with direction from cell  $i$ 
      to cell  $(i - 1)$  */
11:   else
12:     $U := Normalize(P_i - P_{i-1})$ 
13:   end if
14:    $P_{i+1} := P_i + Rand(0, 1) \times s \times RandGauss(U, g(v_i - v_{i-1}))$  /* random
      walk */
15:   if  $num\_of\_computed\_cells \geq c \cdot total\_num\_of\_cells$  then
16:     break
17:   end if
18:    $i := i + 1$ 
19: end loop

```

the future when need arises. The algorithm we proposed, called *directed random walk*, is summarized as follows: The executor first randomly selects k cells that are uniformly distributed in the surface perspective. Then starting from each of the k cells, a *directed random walk* is performed to pick the next cell to compute. The details are sketched in Algorithm 9. Whenever a step is taken, a new cell (P_i) is selected and its value (v_i) gets computed (lines 4-7). The value (v_i), together with the value of the cell where the random step is taken from (i.e. v_{i-1}), is fed into a function to evaluate the interestingness of the region. If the returned value is greater than the threshold (line 9), it means more cells between the two (P_{i-1} and P_i) need to be visited. The direction of the next step of the random walk is therefore set to have a mean U facing towards the previous cell (line 10). Otherwise, the mean direction will be a reversed one (line 12). The actual direction for the next step is determined by function *RandGauss()* which returns a random unit vector following Gaussian distribution with mean U and variance a function of the difference between v_i and v_{i-1} (line 14). The query executor runs k random walks in an interleaving fashion and terminates when user interrupts or a certain percentage (typically less than 50%) of the total output cells have been produced (lines 15-16). At this stage, the users should already have a very good overview of the query results. If the execution has not been terminated, it means a complete and accurate result is needed. In that case, the executor reverts to the strategy described in Section 5.7.2 to compute the remaining output cells.

5.8 Experimental Setup

In this section we describe our prototype system as well as the query and dataset used to experimentally evaluate the HyperGrid model.

5.8.1 Implementation

As an application specifically designed for processing scientific environmental data, we choose to build the system using Mathematica [70], rather than a general-purpose programming tool such as C or Java. There are three main reasons for this choice: Firstly, Mathematica, as an excellent tool for mathematical computations, has the built-in capability to optimize numerical computations, which makes it particularly suitable for processing computationally intensive scientific operations. With Mathematica, we can save efforts from finding the best algorithms for solving particular mathematical problems and focus on the query processing aspect of the system. Secondly, an important feature of Mathematica, which general-purpose programming tools do not provide directly, is the powerful support for symbolic computation. It allows physical models to be manipulated precisely throughout the computation. Finally, we choose Mathematica because it is the tool many scientists often use and hence are already familiar with. So it would be easier for them to maintain and extend the system when necessary in the future.

The system consists of three main components: a query engine, an optimizer and a user interface. The query engine is the core of the system. It executes a query plan according to the given execution strategy. For each operation in the plan, the engine compiles the topology and data specifications given

in the source and target perspectives, and performs the data transformations accordingly when the operation is invoked. The optimizer does a few things: rewrites the query, computes the optimal execution strategy, interacts with the engine to implement the buffering strategy and schedules the output sequence as discussed in Section 5.7. Lastly, a user interface is provided to allow user to specify perspectives, including customized user-defined data and input selection functions. So far, all user-defined functions need to be written in mathematica code. However, with MathLink [71], a generalized application interface provided by Mathematica, we do not see big obstacles to incorporate the current system with user-defined functions in C, Java or other languages.

To evaluate the performance of the HyperGrid system, we also implemented a pure array-based approach using Mathematica for comparison. The array-based approach represents the traditional way in which scientific data is processed to answer a query. There is no integrated query engine which automates the step-by-step data transformations to reach the final answer. Instead, the program offers functions to perform each individual data transformation operation, such as convert, merge and interpolate. Hence, user needs to manually organize the query plan tree. In addition, because an array-based approach only performs computations between arrays without the spatial and temporal context, an external function is required to translate the spatio-temporal points to the corresponding array representation for each data transformation as well as any physical model involved.

Query ID	Query Description
Q1	as Illustrated in Example 5.3.1
Q2	Same as Q1 except the measurement type changes from “ambient temperature” to “watermark”
Q3	Same as Q1 except the clipping window along the time dimension is increased by 100%
Q4	Same as Q1 except the clipping window along latitude and longitude dimensions are increased by $(\sqrt{2} - 1)$ respectively
Q5	as Illustrated in Example 5.9.1

Table 5.3: Query set description

5.8.2 Dataset

The data we use for our experiments were collected from a SensorScope network which was deployed at the Grand-St-Bernard pass in Western Alps at 2400 m in September and October 2007 to monitor the ecological condition of the region. There are totally nine types of meteorologic measurements, namely ambient temperature, surface temperature, humidity, solar radiation, soil moisture, watermark, rain meter, wind speed and direction. Each type of measurements consists of over 588,000 data points. And each data point can be identified by the time and the location where it was measured.

5.8.3 Query Set

We use two categories of queries for our experiments. The first category consists of the query in Example 5.3.1 and its variants (details are described in Table 5.3). They represent routine queries which scientists use frequently to compute statistical information about the data. Typical steps of routine queries include data cleaning (through *convert*), alignment (through *aggregate*), inter-

potation (through *interpolate*) as well as other query-specific operations. The second category simulates the scenario where user wants to explore the data through visualization. We use the query in Example 5.3.2 for our experiment.

5.9 Performance Evaluation

The performance evaluation has two objectives. Firstly, we would like to assess the usability of HyperGrid as a tool to manage scientific environmental data, and compare it against the traditional array-based method. Secondly, we want to evaluate the effectiveness of our proposed optimization strategies in improving the users' experience when they use the system.

All the experiments were conducted on a 2.33 GHz Intel dual core machine with 4 GBs of memory running windows XP.

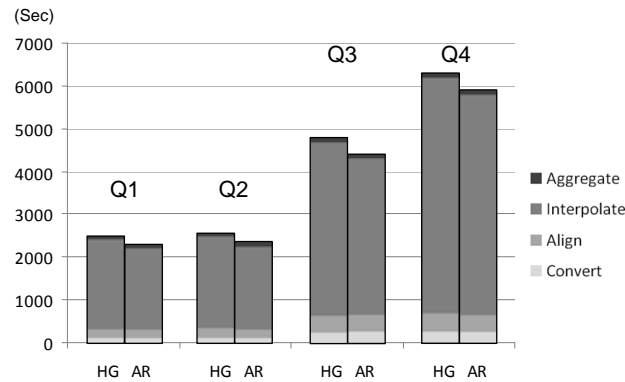


Figure 5.3: HyperGrid (HG) Vs. Array (AR)

5.9.1 Routine Query Execution

Figure 5.3 reports the overall runtime performance of the native HyperGrid (using “bottom-up” strategy without any optimization) implementation and the

array-based implementation for four different routine queries. Q2, Q3 and Q4 are variants of Q1 that vary the workload by either changing the measurement type or the spatio-temporal scope of the *surface*. In all the cases, the queries go through four data transformations (*convert*, *align*, *interpolate* and *aggregate*). From the figure, it is evident that *interpolate* always takes up the majority of the run time. This is because interpolation often comes with expensive user-defined data and input selection functions. In our experiments, we adopt the kriging model for interpolation. It uses an adapted k-nearest neighbor (k-NN) algorithm as the input selection function and a variogram model to estimate the degree of dependence between data points. Typically, some 15 to 22 neighbors are selected and used in the variogram for each target cell to be interpolated. Comparatively, other operations use either the built-in data functions or user-defined functions with much lower complexity, hence they consume significantly less CPU.

5.9.2 HyperGrid vs. Array-based Implementation

The plot corresponding to array-based implementation (referred as “AR”) in Figure 5.3 assumes the ideal scenario that human efforts for writing external functions (for each data transformations in the query plan) to translate spatial and temporal coordinates to the array elements are assumed to be zero. However, in practice this is often a tedious and time-consuming task, which cannot be quantified and reflected in the figure. Even with this unrealistic assumption (by ignoring all hidden costs incurred in array-based processing), HyperGrid (referred as “HG”) only takes slightly longer time (less than 8%)

5.9. PERFORMANCE EVALUATION

than array-based implementation for all the test cases. This indicates that the *HyperGrid* model indeed incurs little overheads to the system. More importantly, by unifying all operations in a standard way and automating the entire query process, HyperGrid can explore optimization opportunities and further boost the runtime performance as illustrated in the next few sections.

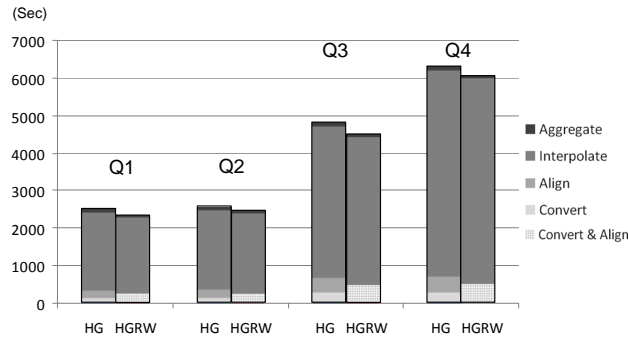


Figure 5.4: Effect of query rewrite

5.9.3 Query Rewrite

Figure 5.4 compares the runtime performance between *HyperGrid* query plans with and without query rewrite. We can see for all the test queries, the *convert* and the *align* perspectives are coalesced after the rewrite. And the rewritten plan (denoted by “HGRW”) clearly runs faster than the original one for all the cases. However, the improvement is not very impressive. This is because the execution cost to compute the *convert* and the *align* perspectives does not constitute a significant portion of the total cost. If we break down the total runtime cost, for example Q1, we can see that the time taken to compute *convert* and the *align* in total is dropped from 341 sec to 245 sec. The saving is actually quite substantial.

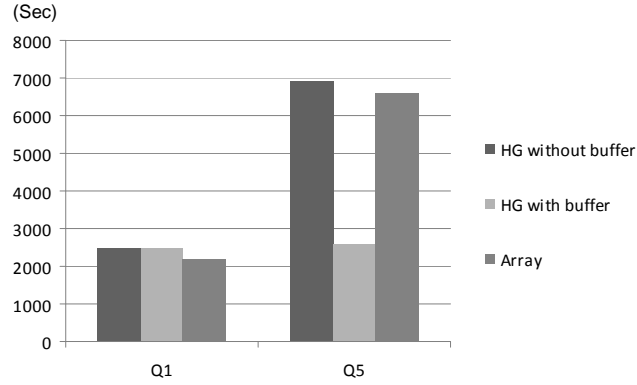


Figure 5.5: Effect of buffer strategy

5.9.4 Buffering Strategy

Next, we evaluate the performance of the buffering strategy proposed in Section 5.7.2. The left part of Figure 5.5 depicts the runtime comparison between HyperGrid with buffering and two other approaches (native HyperGrid and array-based approach) for query Q1. It shows the buffered HyperGrid strategy does not improve the total runtime cost. This is expected because in Q1, all cells in the *surface* do not overlap in the spatio-temporal domain. It means the buffering strategy proposed in Section 5.7.2 would not be beneficial here because no previously computed results are reused. In order to evaluate the proposed buffer strategy for the case where output cells are overlapping in the spatio-temporal domain. We consider a new query variant as follows:

Example 5.9.1 *Return the ambient temperature averaged over 15-minute interval for the period from 2007-10-01 00:00 to 2007-10-04 00:00 and for the region $[45^{\circ}52'1''N, 45^{\circ}52'23''N]$ in latitude and $[7^{\circ}10'37''E, 7^{\circ}10'59''E]$ in longitude on a $1'' \times 1''$ grid. The result should be updated every 5 minutes.*

5.9. PERFORMANCE EVALUATION

The fundamental difference between Q1 and the above query (Q5) is that Q5 averages the data points over 15-minute interval instead of the entire three days. And a new result is generated for every 5-minute advancement along the time dimension. The output of Q5 essentially corresponds to results from a sliding window over time dimension with window size of “15-minute” and sliding step of “5-minute”. This means each pair of the adjacent cells in the *surface* overlaps by “10-minute” on the temporal space. Experimental results of running Q5 is shown on the right of Figure 5.5. As expected, buffered HyperGrid strategy achieves significant runtime reduction this time owing to the effective buffer strategy that avoids doing the redundant computations. On the other hand, the native HyperGrid approach and the array-based method require much more time to process the query due to their inability to recognize and reuse previously computed intermediate results.

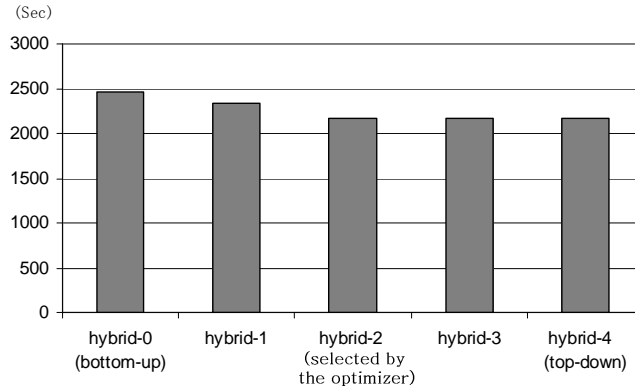


Figure 5.6: Optimizing execution strategy (total runtime)

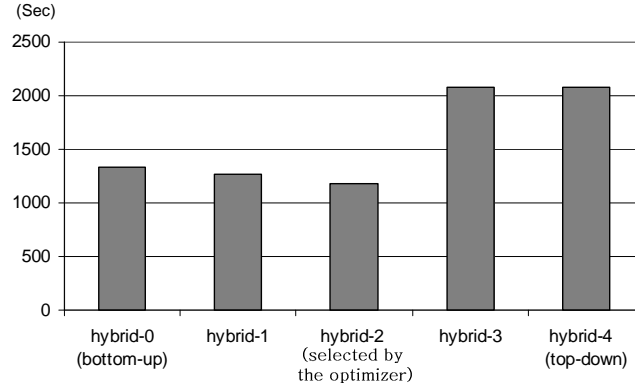


Figure 5.7: Optimizing execution strategy (average response time)

5.9.5 Optimizing Execution Strategy

Figures 5.6 and 5.7 depict the performance of all possible execution strategies for Q1 in terms of total runtime cost and average response time respectively. No other optimization technique, such as query rewrite, is enabled for this test case. So there will be 4 perspectives in the query plan and hence 5 possible execution strategies (from hybrid-0 to hybrid-4). Hybrid-0 is essentially the “bottom-up” strategy and hybrid-4 is the “top-down” strategy. In terms of the total runtime cost (Figure 5.6), hybrid-0 gives the worst performance due to the two reasons explained in section 5.7.2. The total run time gets reduced as more perspectives are computed in a “top-down” manner. It is no surprise that hybrid-4 gives the shortest total runtime. However, in terms of the average response time (which we think is the critical metric), hybrid-4 performs the worst. This is because no output cells are produced until the last perspective (i.e. the *surface*) starts to be computed. We can see from the figure that based on the cost model given in Section 5.7.2, the optimizer successfully finds the optimal execution strategy (in this case, hybrid-2) for the given query.

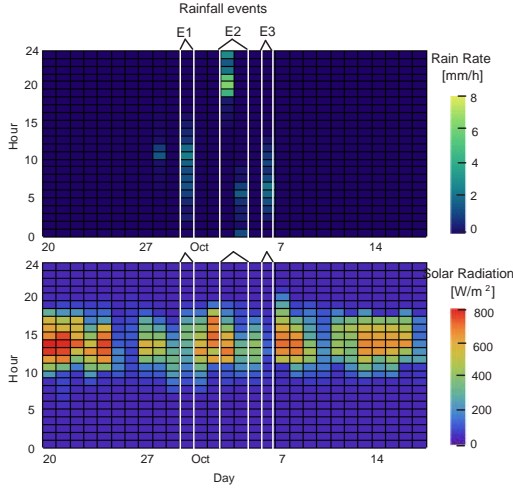


Figure 5.8: Plot with 15% result computed (using directed random walk algorithm)

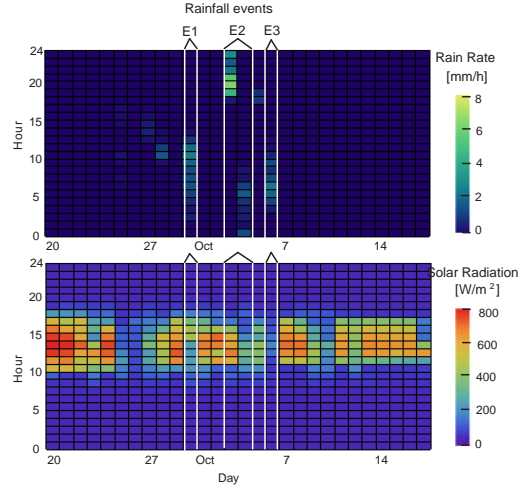


Figure 5.9: Plot with 100% result computed

5.9.6 Visualization Optimization

Lastly, we evaluate the optimization strategy for visualization proposed in Section 5.7.3. We use the scenario in Example 5.3.2 for this experiment. Basically the user needs to visualize the output of the hourly average rainfall rate together with the hourly average solar radiation values to discover whether the two are correlated. Figure 5.8 shows the results produced by the directed random walk approach which only computes 15% of the total output cells. The remaining 85% of the cells in the figure are obtained by applying a simple smoothing function. For comparison, the actual accurate output (with 100% output cells computed from the dataset) is shown in Figure 5.9. In both figures, the graph on the top indicates the hourly average rainfall rate for a period of 28 days (from Sep 20 to Oct 17 as indicated in the horizontal axis). Hour of a day is indicated in the vertical axis. The graph below is the corresponding solar

5.9. PERFORMANCE EVALUATION

radiation values during this period. As we can see, three rainfall events (E1, E2, E3) are identified from the rainfall rate graph. Each event is highlighted by a pair of white vertical bars that run across both graphs. In the solar radiation graph (either the one in Figure 5.8 or Figure 5.9), we can see that the values during the period of the rainfall events (especially E1 and E3) are lower (darker color) than other days for the same hour. This hints that rainfall and solar radiation are very likely to be correlated for this region. By comparing Figure 5.8 and Figure 5.9, we can see that most of the trends or patterns exhibited in Figure 5.9 can also be found in Figure 5.8. This clearly indicates that the directed random walk algorithm does a very good job in simulating the actual results by processing only a small fraction (15% in this example) of the output cells. Figure 5.10 reports the runtime costs for generating the two figures. For both the rainfall and solar radiation datasets, the time required by the directed random walk algorithm to simulate the results is less than 1/5 of the time needed by the cost-based optimization algorithm to compute the full results. This shows the directed random walk algorithm can greatly improve the data exploration efficiency in many instances.

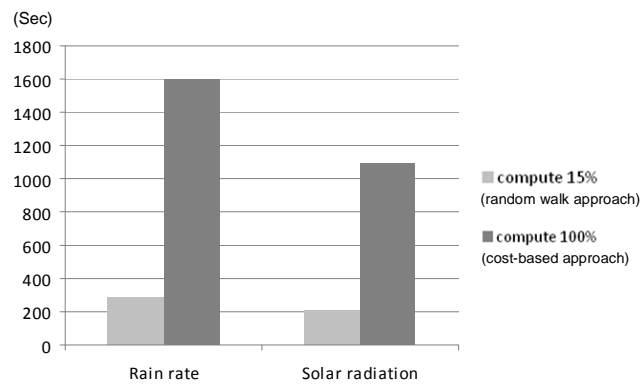


Figure 5.10: Run time cost for visualization

5.10 Data-driven Memory Management for Hyper-Grid

5.10.1 Implementing Dynamic Base Perspectives

In the above experiments, we assume queries only access historical data which are stored as local files. That means all the inputs required to build the base perspective are already available before the queries are issued. This is, however, not applicable to continuous queries where the input involves real-time streams. In that case, the *base* becomes a dynamic data structure which updates as new sensor input flows in.

Note when dealing with real-time streams, many scientific queries impose a strict temporal requirement on the update pattern of the *base*. More specifically, a base perspective often needs to be updated in a monotonically increasing manner on time dimension. As an example, consider a query which monitors temperature variation by comparing the latest temperature value with the one previously reported and alerts if their difference is greater than Δ . The input sequence received by the query processor is crucial to the correctness of the query result. In this case, each involved perspective including the *base* has to be updated in an order determined by the inputs' timestamp values.

Typically, a perspective comprises data from multiple sources. For example, for a query which continuously updates the latest average temperature for a given region R , the inputs involved to build the base perspective embrace measurements from all the sensor stations deployed in that region with each

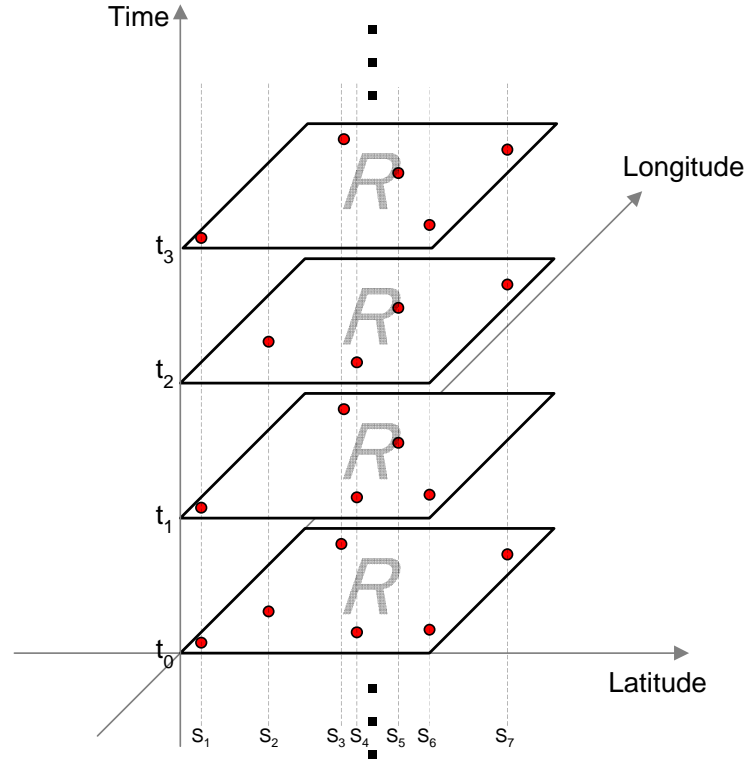


Figure 5.11: Illustration of a dynamic base perspective

station being an independent stream generator. To combine multiple real-time streams to build a base perspective requires an implicit synchronization procedure which joins data from different streams on their timestamp values. This is a necessary step to ensure the *base*'s update follows temporal monotonicity. Figure 5.11 elaborates this in more details. The figure shows a graphical illustration of a dynamic base perspective. In the given scenario the region R includes seven input streams (S_1 to S_7). Each stream aperiodically transmits the latest sensor reading to the central server where the *base* is built. Due to transmission delay and unreliable communication channels, data may not be or-

5.10. DATA-DRIVEN MEMORY MANAGEMENT FOR HYPERGRID

dered according to their timestamp values and there may be lags among inputs from different streams. The synchronization procedure essentially organizes the scrambled data by sorting them in order and joining them together. Each joined result forms a time-slice of the data map for the region R . Figure 5.11 shows four such time-slices (t_0 to t_3). Each contains several data readings with the same timestamp values. A base perspective is updated slice by slice along the time axis. In our example, time-slice t_0 is appended to the *base* first, followed by time-slice t_1 , then t_2 and so on.

5.10.2 Using Data-driven Memory Management

The data-driven memory management scheme discussed in Chapter 3 can be very useful here for building dynamic base perspectives with monotonic temporal update. By exploring properties such as *intra-stream* delay and *inter-stream* delay, the processor is able to perform the data synchronization with much less memory overhead. To verify the effectiveness of data-driven memory management in the context of a HyperGrid system, we set up the following experiment: We use the same input data set as described in Section 5.8.2. But this time, instead of directly consumed by the HyperGrid processor, the inputs are sent to a stream generator, which reproduces the sensor data as streams with customized Scrambling Factor (SF) and *Lag*. The value of SF for each stream is between 0 and 200 and the *Lags* among the streams are between 0 min and 10 min. We compare the memory consumption of producing the dynamic *base* with and without using our data-driven memory management strategy. When the data-driven memory management strategy is not used, a fixed window is

5.11. MULTI-QUERY SCHEDULING IN HYPERGRID

Input Data Scrambling	Average Memory Saving
$SF = 0; Lag = 0$	0%
SF between 0 and 200; $Lag = 0$	46%
$SF = 0; Lag$ between 0 min and 120 min	12%
SF between 0 and 200; Lag between 0 min and 10 min	61%

Table 5.4: Average memory saving with the data-driven memory management scheme

employed for each input to buffer the stream tuples. The window size is set to be just large enough to ensure similar output quality as the data-driven memory management strategy can be produced. Table 5.4 summarizes the results. The table lists on average how many percentages of memory space can be saved by using the data-driven strategy. Evidently, the savings are substantial for disordered or unsynchronized streams.

5.11 Multi-Query Scheduling in HyperGrid

In Section 5.7.2, we discussed several alternative ways to execute a query in HyperGrid (“top-down”, “bottom-up” and “hybrid-k”) and compared their differences. This is in fact an intra-query scheduling issue. When there are multiple queries running concurrently in the system, the issue of inter-query scheduling deserves equal attention. Running scientific queries can be very costly. Hence a smart choice of execution among different queries is crucial when the workload is heavy. This is especially true for applications where prompt delivery of output are vital for some queries.

In Chapter 4, we introduced several multi-query scheduling strategies for data stream applications. Some of them can be readily employed in Hyper-

Grid as well. Compared to general data stream scheduling, some noticeable differences in HyperGrid scheduling are as follows. First, the reevaluation of a continuous query in HyperGrid is triggered by an update of the base perspective. While in general stream scheduling, query evaluation is triggered by a new arriving tuple. This implies that in order to apply our previously proposed job scheduling strategies in HyperGrid, the job triggering event should be the presence of a new time-slice in the base perspective instead of the arrival of a new input tuple. Second, scientific queries may not impose hard deadlines on output delivery. Instead, they require results to be delivered as soon as possible. This means deadline-aware strategies are less appropriate here since their scheduling decisions are heavily influenced by the deadline values.

Hence, in the following experiment we focus on greedy scheduling strategies (specifically, the basic strategy and the OptProfit strategy discussed in Section 4.6) that prioritize queries mainly by their importance and are less sensitive to deadlines. We use the same experimental set up described in Section 5.10.2 for the input data preparation. The queries used in the experiments are generated randomly. Each query contains two to four perspective transformations. Depending on the type of usage, the queries are classified into three categories: alert query, user query and archiving query. Alert queries refer to monitoring queries that alert when unusual patterns or phenomena are detected from the input. They are the most time-critical queries and hence have the highest priority. Each alert query is given a weighting factor of 10. User queries are those created by scientists for data exploration or assertion verification purposes. They have medium priority with weights between 2 and 5. Archiving queries are created for archiving purpose. They are the lowest

5.11. MULTI-QUERY SCHEDULING IN HYPERGRID

priority queries with a weight of 1 each only. All the queries are computed from the same base perspective. Further sharing of perspectives among queries are possible if they involve same intermediate operations. Figure 5.12 shows an example query graph used in the experiment.

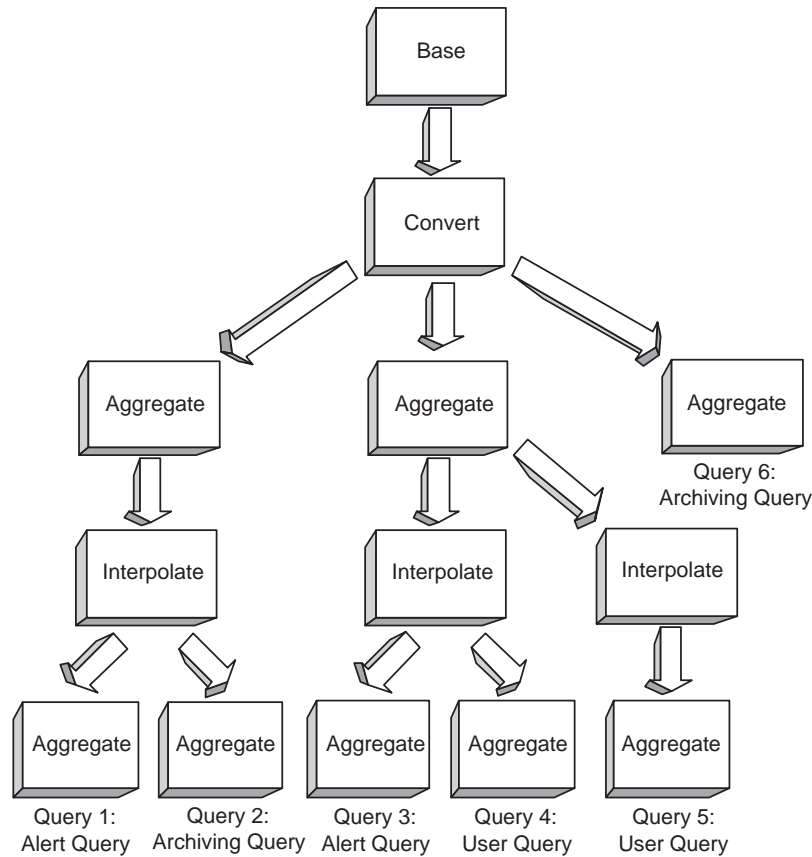


Figure 5.12: An example of multi-query graph

Because there are no specific deadlines specified for the input or queries, any input that is not scheduled in the first instance will not be discarded. Instead, the input will be buffered until being scheduled at a later time. An interesting characteristic to take note here is that all the operations (i.e. perspective transformations) in HyperGrid are idempotent. It means if you feed a HyperGrid

query with the same input twice, the result is not additive. It would be just the same as you feed the input once. This is because the surface perspective (i.e. the output) of a query at time t is only relevant to the content of the input that is available at t , regardless how many times the query has been executed before. For this reason, an input that has not been processed by a query only causes the query's output temporarily outdated. As long as the query is rescheduled in the future, the results will be correct or up-to-date at that time, no matter whether that input has been processed or not. Because of these, the previous QoS score function used in Chapter 4 is not applicable here. A better metric to evaluate the quality of a HyperGrid query would be how frequent its surface perspective is updated. Hence the new score function is designed as follows: When there is a new time-slice available at the *base*, all the queries that need to be updated have the potential to receive a token, say T . The token value is proportional to query's weight. If the query is reevaluated in response to the new time-slice, the token will be credited. Otherwise, the potential token is saved. But its value is depreciated, determined by a decaying factor λ . So when the next update of the *base* occurs, queries that have not been updated previously have the potential to receive a token worth $T + \lambda T$, so on and so forth. In the experiment, the value of λ is set at 0.9.

Figure 5.13 reports the average update frequency of our experimental queries using different scheduling strategies. We can see that OptProfit strategy produces higher update frequency for alert queries and lower update frequency for user and archiving queries. Although Basic strategy exhibits the same trend, the contrast among the three types of queries is not as big as OptProfit. While for Round-robin strategy, the update frequencies of the three types of queries

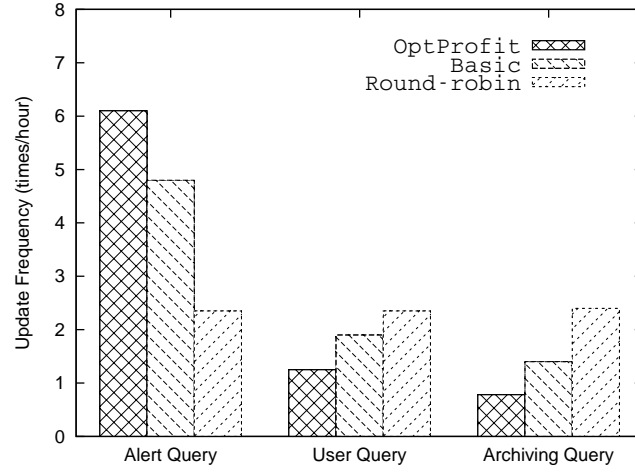


Figure 5.13: Comparison of update frequency

are almost same. So the question is which strategy performs better. The answer is to multiply each query's update frequency with its corresponding weight and add them together. Figure 5.14 plots the result. We can see that OptProfit has the highest total weighted update frequency. It means the strategy has the best resource distribution among different types of queries which leads to the maximal total utility.

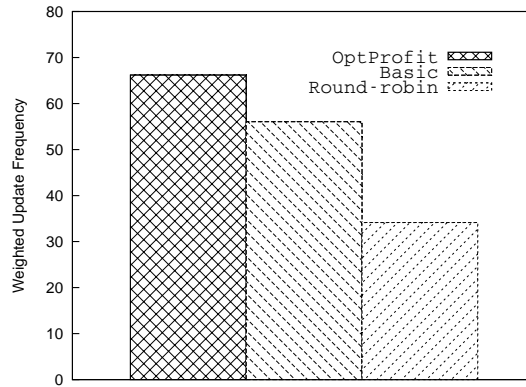


Figure 5.14: Comparison of weighted update frequency

5.12 Summary

We have presented the framework of HyperGrid in this chapter. One objective of this work is to demonstrate that scientific data management can benefit from database technology that enables the integration of scientific workflows, which was largely segregated traditionally. With a uniform data model and database-style processing paradigm, scientific computations can be carried out in a more systematic way. An integrated architecture also reveals opportunities for query optimization. In the second part of the chapter, we discussed how memory management scheme and stream scheduling strategies discussed in the previous chapters can be applied in the context of a HyperGrid environment. It shows these schemes or strategies are practically useful for real data stream applications.

6

Conclusions

Living in the information age with huge amount of data being generated, processed and stored every day, we are relying more than ever on database technology to help us efficiently manage various types of information. The emerging demand for processing data in the form of streams, however, challenges the traditional database processing paradigm. Compared to store based data which are static and predictable, the mass, rapid and unpredictable nature of streaming data calls for more advanced techniques for processing queries efficiently. In this thesis, we attempt to explore the issue from an interesting perspective. By focusing on *time*, the key aspect that distinguishes stream query processing from traditional query processing, we propose several data management techniques that help boost the performance of a stream query processor. Summary of our contributions is given below.

6.1 Summary of Contributions

In the first piece of work, we focus on the memory management issue in data stream processing. Traditionally, the memory requirement for a stream join is *query-driven*. We find that such a *query-driven* approach not only leads to extravagant memory overheads but also produces unsatisfactory query answers. Hence, we introduce the concept of *data-driven* memory management and contend that, whenever possible, memory allocation for stream join should be *data-driven* instead of *query-driven*. Following this, we propose a new stream join processing scheme, called *Window-Oblivious Join* (WO-Join), which exploits the inherent notion of time associated with input tuple so that a system can dynamically adjust the memory buffer size to minimize unnecessary memory overheads. Our experiments suggest that WO-Join significantly outperforms traditional windowed join in terms of both output quality and memory efficiency.

In many data stream applications, overwhelming streaming input could easily overload the query processor. When this occurs, the system should intelligently allocate the limited resource among queries of different urgency and importance so that the total QoS of the system can be maximized. In the second piece of work, we focus on real-time query scheduling techniques for achieving the goal. We point out that the traditional operator-based scheduling strategies are insufficient to address issues arising from the real-time requirements of output generation in DSMS. What is needed is a fine-grained resource control scheme that works at the tuple level. Inspired by the classic job scheduling algorithms, we propose several tuple-based stream scheduling strategies. These

strategies open a new avenue for addressing stream scheduling issues. And the effectiveness of these strategies is verified under different workloads and query settings.

In addition to theoretical studies, we also emphasize practical implementations related to data stream applications. In the last piece of work, we develop a data processing platform for a scientific sensor data application. The main challenge is how to integrate data streams collected from heterogeneous sensor stations and offer a unified data platform to query, analyze and visualize sensor information to facilitate scientific research. We also discuss how to apply the data-driven memory management scheme and stream scheduling strategies covered by our previous works in the context of scientific sensor data processing. This not only helps us understand these techniques better but also justifies their usefulness in practical applications.

6.2 Future Work

Other than what have been studied in this thesis, we believe there are a lot more other time-related data stream issues to be explored. The followings are some of the promising directions for future work.

- **Uncertain data.** Streaming data are often incomplete or contain uncertain information. When the time information associated with stream's input involves uncertainties, a direct application of the memory management scheme or the scheduling strategies will encounter difficulties. Uncertain data may be modeled in different ways such as using probabilistic information or range values. Efforts can be made to extend the techniques

discussed in this thesis to support streams with various uncertain time information.

- **Data lineage.** Time information also plays an important role in data lineage. For example, in scientific domains, researchers may need the chronological relationships of the contributing inputs in order to understand query results better. Unlike some other types of provenance information which may be derived directly through *inversion* by tracing the query graph backward, a chronological history of how data are evolved is not recoverable unless it is explicitly recorded. This is because temporal information requires tuple-level granularity. As far as we know, there is no good annotation scheme proposed for recording the chronology of data at tuple level. It would be interesting to explore innovative techniques that would achieve this in an efficient way.
- **Distributed stream processing.** Distributed query processing constitutes an important part of data stream research. This is mainly for two reasons. First, many input streams are physically distributed. Shipping all the data to a central processor may be too costly. Second, a good distributed processing paradigm improves system's scalability, which is quite important since large scale data stream processing is becoming increasingly popular. Time issues in distributed stream processing is a broad topic with a lot of interesting problems to study. For example, given that the input of a continuous query is distributed to multiple nodes for processing, one topic is to ensure the final result, which combines the sub-result from each node, still observes certain temporal order as if the input

is processed by a single server in a FIFO manner. This can be difficult to achieve for stateful operations such as join and aggregate. Distributed query scheduling is also an interesting topic. When stream queries impose stringent requirements on the timeliness of output delivery, to design an efficient distributed query scheduler can be very challenging, considering the unpredictable communication delay and the synchronization issues among the working nodes.

- [1] Emerging wireless sensor network applications report.
<http://www.bharatbook.com/Market-Research-Reports/Emerging-Wireless-Sensor-Network-Applications.html>.
- [2] Internet traffic archive. <http://www.sigcomm.org/ITA>.
- [3] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
- [4] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *ICDE*, pages 232–243, 1997.
- [5] Mohammed Al-Kateb, Byung Suk Lee, and Xiaoyang Sean Wang. Reservoir sampling over memory-limited stream joins. In *SSDBM*, page 23, 2007.
- [6] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. *Data-Stream Management: Processing High-Speed Data Streams*, chapter STREAM: The Stanford Data Stream Management System. Springer-Verlag, New York, 2005.
- [7] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Trans. Database Syst.*, 29:162–194, 2004.

- [8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. Stanford University Technical Report, 2003.
- [9] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296, 2004.
- [10] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.
- [11] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *SIGMOD Conference*, pages 253–264, 2003.
- [12] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.
- [13] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418, 2004.
- [14] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.
- [15] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k -constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

- [16] Paramvir Bahl and Venkata N. Padmanabhan. Radar: An in-building rf-based user location and tracking system. In *INFOCOM*, pages 775–784, 2000.
- [17] Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eduardo F. Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stanley B. Zdonik. Retrospective on aurora. *VLDB J.*, 13(4):370–383, 2004.
- [18] P. Baptiste. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, 2:245–252, 1999.
- [19] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, Martin Vetterli, Olivier Couach, and Marc Parlange. Sensorscope: Out-of-the-box environmental monitoring. In *IPSN*, pages 332–343, 2008.
- [20] Sanjoy K. Baruah, Gilad Koren, D. Mao, Bhubaneswar Mishra, Arvind Raghunathan, Louis E. Rosier, Dennis Shasha, and Fuxing Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992.
- [21] Peter Baumann. Management of multidimensional discrete data. *VLDB J.*, 3(4):401–444, 1994.
- [22] Peter Baumann, Paula Furtado, Roland Ritsch, and Norbert Widmann. Geo/environmental and medical data management in the rasdaman system. In *VLDB*, pages 548–552, 1997.

- [23] Michael Cammert, Jurgen Kramer, Bernhard Seeger, and Sonny Vaupel. An approach to adaptive memory management in data stream systems. University of Marburg Technical Report No. 49, 2005.
- [24] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.
- [25] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.
- [26] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
- [27] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaraqc: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.
- [28] Min Chen, Richard H. Clayton, Arun V. Holden, and J. V. Tucker. Visualising cardiac anatomy using constructive volume geometry. In *FIMH*, pages 30–38, 2003.

- [29] Min Chen, Andrew S. Winter, David Rodgman, and Steve Treuvett. Enriching volume modelling with scalar fields. In *Data Visualization: The State of the Art*, pages 345–362, 2003.
- [30] Zhimin Chen and Vivek R. Narasayya. Efficient computation of multiple group by queries. In *SIGMOD Conference*, pages 263–274, 2005.
- [31] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD Conference*, pages 150–160, 1994.
- [32] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD Conference*, pages 647–651, 2003.
- [33] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD Conference*, pages 40–51, 2003.
- [34] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.
- [35] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.
- [36] Amol Deshpande and Samuel Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD Conference*, pages 73–84, 2006.

- [37] Luping Ding, Nishant Mehta, Elke A. Rundensteiner, and George T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, 2004.
- [38] Michael J. Franklin, Sailesh Krishnamurthy, Neil Conway, Alan Li, Alex Russakovsky, and Neil Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.
- [39] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *CIKM Conference*, 2005.
- [40] G. V. Gens and E. V. Levner. Fast approximation algorithm for job sequencing with deadlines. *Discrete Applied Mathematics*, 3:313–318, 1981.
- [41] Robert Givan, Edwin K. P. Chong, and Hyeong Soo Chang. Scheduling multiclass packet streams to minimize weighted loss. *Queueing Syst.*, 41(3):241–270, 2002.
- [42] Lukasz Golab, Shaveen Garg, and M. Tamer Özsu. On indexing sliding windows over online data streams. In *EDBT*, pages 712–729, 2004.
- [43] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.
- [44] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *SIGMOD Conference*, pages 658–669, 2005.
- [45] Sudipto Guha and Nick Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *ICDE*, pages 567–, 2002.

- [46] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD Conference*, pages 287–298, 1999.
- [47] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *SSDBM*, pages 75–84, 2003.
- [48] Joseph M. Hellerstein. Online processing redux. *IEEE Data Eng. Bull.*, 20(3):20–29, 1997.
- [49] Bill Howe and David Maier. Algebraic manipulation of scientific datasets. In *VLDB*, pages 924–935, 2004.
- [50] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. In *VLDB*, pages 103–114, 1992.
- [51] Milena Ivanova and Tore Risch. Customizable parallel execution of scientific stream queries. In *VLDB*, pages 157–168, 2005.
- [52] Qingchun Jiang and Sharma Chakravarthy. Scheduling strategies for processing continuous queries over streams. In *BNCOD*, pages 16–30, 2004.
- [53] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A heartbeat mechanism and its application in gigascope. In *VLDB*, pages 1079–1088, 2005.
- [54] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD Conference*, pages 106–117, 1998.

- [55] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.
- [56] Richard Manning Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum, New York, 1972.
- [57] E. L. Lawler. Sequencing to minimize the weighted number of tardy jobs. *RAIRO Operations Research*, 10:27–33, 1976.
- [58] E. L. Lawler and J. Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16:77–84, 1969.
- [59] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, 1994.
- [60] Feifei Li, Ching Chang, George Kollios, and Azer Bestavros. Characterizing and exploiting reference locality in data stream applications. In *ICDE*, page 81, 2006.
- [61] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.
- [62] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conference*, pages 311–322, 2005.

- [63] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *SIGMOD Conference*, pages 228–239, 1996.
- [64] Bin Liu, Yali Zhu, and Elke A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD Conference*, pages 347–358, 2006.
- [65] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. Olap on sequence data. In *SIGMOD Conference*, pages 649–660, 2008.
- [66] Carey Douglass Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986.
- [67] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002.
- [68] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.
- [69] Arunprasad P. Marathe and Kenneth Salem. Query processing techniques for arrays. *VLDB J.*, 11(1):68–91, 2002.
- [70] The wolfram mathematica homepage. <http://www.wolfram.com/>, 2008.
- [71] The mathematica mathlink. <http://www.wolfram.com/solutions/mathlink/mathlink.html>, 2008.

- [72] J. Moore. An n job one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15(1):102–109, 1968.
- [73] Stratos Papadomanolakis, Anastassia Ailamaki, Julio C. López, Tiankai Tu, David R. O’Hallaron, and Gerd Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD Conference*, pages 551–562, 2006.
- [74] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Trans. Netw.*, 3(3):226–244, 1995.
- [75] Laurent Péridy, Eric Pinson, and David Rivreau. Using short-term memory to minimize the weighted number of late jobs on a single machine. *European Journal of Operational Research*, 148(3):591–603, 2003.
- [76] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in exodus. In *SIGMOD Conference*, pages 208–219, 1987.
- [77] Sartaj Sahni. Algorithms for scheduling independent tasks. *J. ACM*, 23(1):116–127, 1976.
- [78] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, 2006.
- [79] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *SIGMOD Conference*, pages 430–441, 1994.
- [80] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Seq: A model for sequence databases. In *ICDE*, pages 232–239, 1995.

- [81] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.
- [82] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Efficient scheduling of heterogeneous continuous queries. In *VLDB*, pages 511–522, 2006.
- [83] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL*, pages 336–343, 1996.
- [84] Ben Shneiderman. Extreme visualization: squeezing a billion records into a million pixels. In *SIGMOD Conference*, pages 3–12, 2008.
- [85] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.
- [86] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.
- [87] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX*, 1998.
- [88] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
- [89] Nesime Tatbul and Stanley B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.

- [90] Tri Minh Tran and Byung Suk Lee. Transformation of continuous aggregation join queries over data streams. In *SSTD*, pages 330–347, 2007.
- [91] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):555–568, 2003.
- [92] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
- [93] Tolga Urhan and Michael J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *VLDB*, pages 501–510, 2001.
- [94] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*, pages 37–48, 2002.
- [95] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.
- [96] Mengzhi Wang, Ngai Hang Chan, Spiros Papadimitriou, Christos Faloutsos, and Tara M. Madhyastha. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *ICDE*, pages 507–516, 2002.
- [97] Tianqiu Wang, Simone Santini, and Amarnath Gupta. An interpolated volume model for databases. In *ER*, pages 335–348, 2003.

- [98] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, 1991.
- [99] Ji Wu, Kian-Lee Tan, and Yongluan Zhou. A real-time system approach to multi-query data stream scheduling. Submitted for Publication.
- [100] Ji Wu, Kian-Lee Tan, and Yongluan Zhou. Window-oblivious join: A data-driven memory management scheme for stream join. In *SSDBM Conference*, 2007.
- [101] Ji Wu, Kian-Lee Tan, and Yongluan Zhou. Data-driven memory management for stream join. *Inf. Syst.*, 34(4-5):454–467, 2009.
- [102] Ji Wu, Kian-Lee Tan, and Yongluan Zhou. Qos-oriented multi-query scheduling over data streams. In *DASFAA*, pages 215–229, 2009.
- [103] Ji Wu, Yongluan Zhou, Karl Aberer, and Kian-Lee Tan. Towards integrated and efficient scientific sensor data processing: a database approach. In *EDBT*, pages 922–933, 2009.
- [104] Junyi Xie, Jun Yang, and Yuguo Chen. On joining and caching stochastic streams. In *SIGMOD Conference*, pages 359–370, 2005.
- [105] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD Conference*, pages 431–442, 2004.
- [106] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369, 2002.