TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science
Systems Engineering Group

Handling Tradeoffs between Performance and Query-Result Quality in Data Stream Processing

Dissertation
zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technische Universität Dresden
Fakultät Informatik

eingereicht von
M.Sc. Yuanzhen Ji
geboren am 27.08.1985 in Longkou, China

Gutachter:

Prof. Dr. (PhD) Christof Fetzer
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Systems Engineering
01062 Dresden, Deutschland

Prof. Dr. (PhD) Pascal Felber
Université de Neuchâtel
Institut d'informatique
Complex Systems Group
CH-2000 Neuchâtel, Schweiz

Tag der Verteidigung: 28. November 2017

Dresden, im April, 2018

Abstract

Data streams in the form of potentially unbounded sequences of tuples arise naturally in a large variety of domains including finance markets, sensor networks, social media, and network traffic management. The increasing number of applications that require processing data streams with high throughput and low latency have promoted the development of data stream processing systems (DSPS). A DSPS processes data streams with continuous queries, which are issued once and return query results to users continuously as new tuples arrive.

For stream-based applications, both the query-execution performance (in terms of, e.g., throughput and end-to-end latency) and the quality of produced query results (in terms of, e.g., accuracy and completeness) are important. However, a DSPS often needs to make tradeoffs between these two requirements, either because of the data imperfection within the streams, or because of the limited computation capacity of the DSPS itself. Performance versus result-quality tradeoffs caused by data imperfection are inevitable, because the quality of the incoming data is beyond the control of a DSPS, whereas tradeoffs caused by system limitations can be alleviated—even erased—by enhancing the DSPS itself.

This dissertation seeks to advance the state of the art on handling the performance versus result-quality tradeoffs in data stream processing caused by the above two aspects of reasons. For tradeoffs caused by data imperfection, this dissertation focuses on the typical data-imperfection problem of stream disorder and proposes the concept of quality-driven disorder handling (QDDH). QDDH enables a DSPS to make flexible and user-configurable tradeoffs between the end-to-end latency and the query-result quality when dealing with stream disorder. Moreover, compared to existing disorder handling approaches, QDDH can significantly reduce the end-to-end latency, and at the same time provide users with desired query-result quality. In this dissertation, a generic buffer-based QDDH framework and three instantiations of the generic framework for distinct query types are presented. For tradeoffs caused by system limitations, this dissertation proposes a system-enhancement approach that combines the row-oriented and the column-oriented data layout and processing techniques in data stream processing to improve the throughput. To fully exploit the potential of such hybrid execution of continuous queries, a static, cost-based query optimizer is introduced. The optimizer works at the operator level and takes the unique property of execution plans of continuous queries—feasibility—into account.

Acknowledgement

First and foremost, I would like to express my sincere gratitude to my advisor, Prof. Christof Fetzer, for providing me the opportunity to write my PhD dissertation in the Systems Engineering Group, and for supporting me whenever I needed during the last years. Second, I would like to thank Prof. Pascal Felber for his commitment to serve as a secondary reviewer. Thirdly, I am also grateful to my former manager at SAP Dresden, Gregor Hackenbroich, for giving me the opportunity to join the industrial PhD program at SAP Dresden, for guaranteeing the time that I needed for my PhD work, and for all the review and feedback on my publications.

My special thanks go to Zbigniew Jerzak and Anisoara Nica. Zbigniew took the responsibility of being my advisor on the company side. He taught me all the essential skills to accomplish such a difficult project, protected me from being too much disturbed by the company work that is irreverent to my PhD project, gave me uncountable suggestions on my research, and introduced me to Ani, another great motivator and driver of my PhD project. Ani supported me throughout the project not only with her expertise, her constructive suggestions and feedback, but also with her passion on research and her consistently positive and optimistic attitude.

I also thank my coauthors Hongjin Zhou and Jun Sun, who worked with me on the topics of quality-driven disorder handling for individual sliding-window aggregate queries and quality-driven disorder handling for individual sliding-window join queries, respectively. They contributed a lot to the initial implementation of the prototype system.

Furthermore, I would like to thank my former or present colleagues at SAP, especially Thomas Heinze, Uwe Jugel, and Elena Vasilyeva, and other PhD students in the Systems Engineering Group for all the inspirations and discussions.

Last but not least, I am very thankful to my husband and my parents for their continuous motivation and support.

Without any of you, it would not be possible to accomplish this PhD project. Thank you!

Publications

The content of this dissertation is based on the following peer-reviewed publications:

[Ji+16a] Yuanzhen Ji, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. "Quality-driven disorder handling for concurrent queries with shared operators". In: *Proceedings of the 10th ACM International Conference on Distributed Event-Based Systems*. DEBS '16. ACM, 2016, pp. 68–79

[Ji+16b] Yuanzhen Ji, Jun Sun, Anisoara Nica, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. "Quality-driven disorder handling for m-way sliding window stream joins". In: *Proceedings of the 32th IEEE International Conference on Data Engineering*. ICDE '16. IEEE, 2016, pp. 493–504

[Ji+15a] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. "Quality-driven processing of sliding window aggregates over out-of-order data streams". In: *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. DEBS '15. ACM, 2015, pp. 25–36

[Ji+15b] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. "Quality-driven continuous query execution over out-of-order data streams". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015, pp. 889–894

[Ji+15c] Yuanzhen Ji, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. "Optimization of continuous queries in federated database and stream processing systems". In: *Proceedings of the 16th Conference on Datenbanksysteme für Business, Technologie und Web (BTW)*. 2015, pp. 403–422

[Ji13] Yuanzhen Ji. "Database support for processing complex aggregate queries over data streams". In: *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. EDBT '13. ACM, 2013, pp. 31–37

Contents

List of Figures

List of Tables

List of Algorithms

1
Introduction

Driven by the expanding coverage of interconnected devices such as sensors, mobiles, and computers, as well as the growing popularity of the World Wide Web, the past two decades have witnessed an increasing number of applications that require processing information appearing in the form of potentially unbounded sequences of continuously arriving data items—termed *data streams*—in (soft) real time. Examples of such stream-based applications can be found in a large variety of domains including finance [ZS02; CDN11; CR13], sensor networks [Jer+12; MZJ13; JZ14], social media [Sha11; Alv+12; Tos+14], and network traffic management [BW01; Cra+02; QMF13]. For instance, people may wish to analyze the financial data coming from stock markets and news feeds to conduct electronic trading; a power station may wish to analyze the energy consumption data reported by smart meters to dynamically adjust the power generation rate; a social-networking service like Twitter may wish to detect the trend of users' conversations; and an internet service provider may wish to monitor the network traffic to detect critical situations such as congestion and denial of service.

The "continuous" nature of data streams determines that, in order to reflect the information carried by newly arrived data items in the results of an analysis performed over the data streams in real time, the analysis itself needs to be continuous as well. Conventional data management techniques used by database management systems (DBMS) cannot provide adequate support for this class of *stream-based applications*. Hence, new data management techniques have been developed, under the banner of *Data Stream Processing* (DSP). Systems designed for processing data streams are, in general, referred to as *data stream processing systems* (DSPS). A DSPS performs analyses over data streams in the form of *continuous queries* [Ter+92; LPT99], which are issued once and return query results incrementally as new data items arrive.

1.1 Tradeoffs between Performance and Query-Result Quality

For stream-based applications, both the performance of the query execution and the quality of produced query results are important. The query-execution performance refers to, for instance, the throughput and the end-to-end latency; and the query-

result quality refers to, for instance, the accuracy, the completeness, and the order of the produced results. Stonebraker et al. [Sto+05] have argued that a DSPS must "*have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications*", and "*guarantee predictable and repeatable outcomes*".

It would be ideal if a DSPS could provide both high query-evaluation performance and high query-result quality. However, in practice, a DSPS often needs to make tradeoffs between these two requirements, either because the data being processed is imperfect, or because the system itself has limited computation capability. For example, in the data aspect, a typical case of data imperfection faced by a DSPS is the disorder of data items within streams [GÖ03a; Sto+05]. Namely, the order in which data items arrive at the system may be different from the order in which they were generated at external data sources (e.g., sensors). To guarantee a high result quality of queries that involve order-sensitive operators, additional effort, thus computation resources, need to be taken to handle the stream disorder. Moreover, the release of query results may need to be delayed to wait for the data items that arrive late, which increases the end-to-end latency of the query processing. In the system aspect, a DSPS may have insufficient computation resources or an inefficient system implementation. If the capacity of a system cannot match the workload posed by queries submitted to the system, the system needs to apply techniques such as load shedding to prevent system overload and severe performance degradation. However, load shedding would impair the query-result quality.

The performance versus result-quality tradeoffs caused by data imperfection are inevitable, because a DSPS has no control over the quality (in terms of, e.g., the arriving order or the completeness) of the data items arriving at the system. In contrast, it is possible to reduce—even erase—the tradeoffs caused by system limitations, by improving the system itself.

1.2 Research Questions and Contributions

This dissertation studies *how to deal with tradeoffs between the performance and the query-result quality in data stream processing*. Specifically, for tradeoffs caused by data imperfection, because these tradeoffs are inevitable and different applications often prefer different points in the spectrum of a tradeoff, this dissertation studies *how to provide flexible and user-configurable tradeoffs*. In particular, the ubiquitous stream-disorder problem is taken as a representative case of data imperfection in the study of this research question. For tradeoffs caused by system limitations, this dissertation studies *how to enhance a DSPS to reduce such tradeoffs*. Although a DSPS can be enhanced in many different ways and a great deal of proposals have been made since the emergence of the first generation of DSPSs, this dissertation seeks solutions in the direction of leveraging different data management techniques for hybrid execution of continuous queries, which has been explored only to a lesser degree.

Corresponding to the research questions defined above, this dissertation makes the following contributions:

- To provide a flexible and user-configurable performance versus result-quality tradeoff when dealing with stream disorder, this dissertation proposes the concept of *quality-driven disorder handling* (QDDH), along with a generic framework that implements this concept. The key performance metric influenced

by disorder handling is the end-to-end latency. The proposed QDDH concept complements state-of-the-art disorder handling techniques and allows minimizing the end-to-end latency while honoring the user-specified requirements on the query-result quality. Because the specific metric for measuring the result quality of a continuous query depends on the type of the query, this dissertation studies instantiations of the generic QDDH framework for two types of continuous queries—sliding-window aggregate (SWA) queries and *m*-way sliding-window join (MSWJ) queries, which are at the heart of many stream-based applications. For each query type, components of the generic QDDH framework are instantiated in specific ways to address the particular semantics of the query type.

- The concept of QDDH can be applied on top of different existing disorder handling techniques such as the buffer-based technique [Aba+03; BSW04] and the punctuation-based technique [SW04a; Li+08]. To support data streams in the most generic form, the buffer-based technique is chosen in this dissertation. As a result, the objective of QDDH boils down to dynamically adjust the sizes of the buffers used for disorder handling, so that the end-to-end latency is minimized and at the same time the user-specified result-quality requirement is satisfied. To this end, an analytical model is proposed, which captures the relation between the applied buffer sizes and the consequent query-result quality directly. Compared with modeling methods that treat this relation as a black box, the proposed analytical model allows searching for the optimal buffer sizes to meet the user-specified result-quality requirement in each iteration of the buffer-size adaptation.

- Based on the instantiations of the QDDH concept for individual queries, this dissertation continues to study how to apply QDDH for concurrent queries that share query operators. When an operator is shared across multiple queries, the disorder handling of the output stream of this operator can be shared as well, which, potentially, can reduce the memory cost incurred by disorder handling. This dissertation proposes the notion of chained disorder-handing buffers, which allows sharing the disorder-handling effort at query operators that are shared by multiple queries with different result-quality requirements, to achieve the objective of QDDH for each query. Moreover, two algorithms are proposed to determine the configuration of disorder-handling buffers within an entire query-operator network of concurrent queries. Both algorithms have a linear time complexity. One of them can achieve the objective of QDDH with a minimum memory consumption, and the other trades the memory optimality for low computational cost.

- To improve the performance of a DSPS, following the philosophy that "no one size fits all" [LHB13], this dissertation explores the potential of leveraging both the row-oriented data-processing technique and the column-oriented data-processing technique to process data streams. As a proof of concept, a prototype system that consists of a row-oriented DSPS and a column-oriented in-memory database is built for hybrid execution of continuous queries. To achieve the best query-execution performance, a static, cost-based optimizer is proposed to

optimize select-project-join-aggregate (SPJA) continuous queries. Given a continuous query, the proposed optimizer uses characteristics of the query and the input data streams to determine the optimal placement of each operator of the query within the hybrid system. This fine level of optimization, combined with the estimation of the feasibility of query execution plans, allows the optimizer to devise hybrid query execution plans that achieve a throughput that cannot be matched by either component system alone.

1.3 Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter 2 provides more insight into data stream processing, introduces the semantics of continuous-query execution that is adopted in this dissertation, and presents related work in both the area of handling data imperfection within data streams and the area of enhancing DSPSs for higher system performance.

Chapter 3, 4, and 5 present contributions of this dissertation in the area of handling the tradeoff caused by a representative case of data imperfection—stream disorder. Specifically, Chapter 3 motivates the idea of QDDH, introduces the buffer-based disorder handling approach that this dissertation has chosen as the basis for applying QDDH on top, and describes the generic buffer-based QDDH framework as well as the theoretical foundation of the analytical-model-based buffer-size adaptation method that this dissertation proposes to perform quality-driven buffer-size adaptation at the query runtime. The core of this analytical-model-based buffer-size adaptation method is to directly model the relation between the sizes of the applied disorder handling buffers and the consequent query-result quality. For the purpose of comparison, it is also described in this chapter a buffer-size adaptation method that is based on the usage of a *proportional-derivative* (PD) controller [Lev11]. This PD-controller-based buffer-size adaptation method essentially treats the relation between the applied buffer sizes and the query-result quality as a black box.

Chapter 4 describes two instantiations of the generic QDDH framework for two representative types of continuous queries in stream-based applications: SWA queries and MSWJ queries. For each query type, it is described in detail the adopted result-quality metric and the analytical-model-based and the PD-controller-based buffer-size adaptation methods tailored for the particular query type.

Based on the results presented in Chapter 4, Chapter 5 takes one step further to study how to apply QDDH for concurrent SWA and MSWJ queries that have shared source and filter operators.

Chapter 6 presents the contribution of this dissertation in the area of enhancing a DSPS to reduce the performance versus result-quality tradeoffs caused by system limitations. It describes a prototype system, which consists of a row-oriented DSPS and a column-oriented in-memory database, for exploiting the potential of combining the row-oriented and the column-oriented data layout and processing techniques in processing data streams. It also introduces a cost-based query optimizer for optimizing relational SPJA continuous queries in such a hybrid system.

Finally, Chapter 7 concludes this dissertation and discusses possible areas for future work.

All notations used in this dissertation are summarized in a table at the end of the dissertation (cf. Symbols).

2
Background

This chapter first introduces basic concepts related to data stream processing, which form the foundation of this dissertation. It then provides an overview of existing techniques for dealing with the two major factors, i.e., data imperfection and system limitation, that cause tradeoffs between the performance and the query-result quality in data stream processing. This chapter reviews related work only at a high level; related work for the specific techniques proposed in this dissertation is covered in respective chapters.

2.1 Data Stream Processing

The abstract architecture of a DSPS is described in Figure 2.1. Stream-based applications describe stream-analyzing tasks using continuous queries, which are deployed and executed in a DSPS. A continuous query, or a query for short, takes one or several data streams originating from external data sources as input, and produces query results in the form of data streams as well. Examples of data sources of streams include sensors, web applications, and stock exchanges. Result streams of continuous queries are returned back to the applications; and in addition, they can be persisted in storage systems such as database systems for future on-demand analysis.

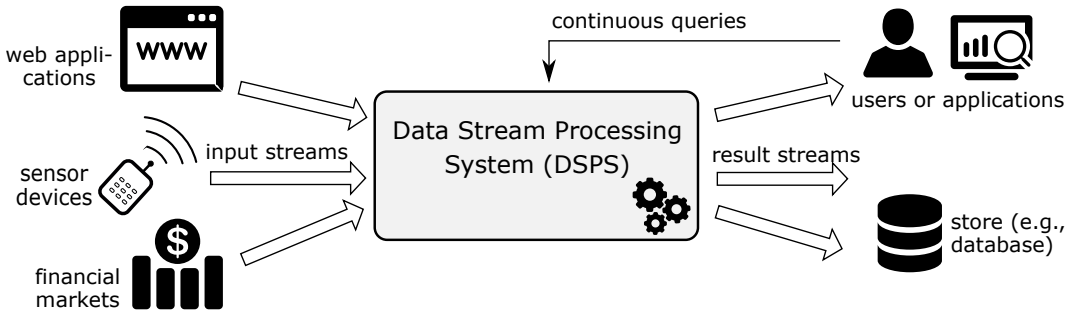


Figure 2.1: Abstract architecture of a DSPS

The history of continuous queries dates back to the early 1990s, when they were introduced for the first time in the Tapestry system [Ter+92]. Tapestry was built on top of an append-only relational database. It converts a continuous query into an in-

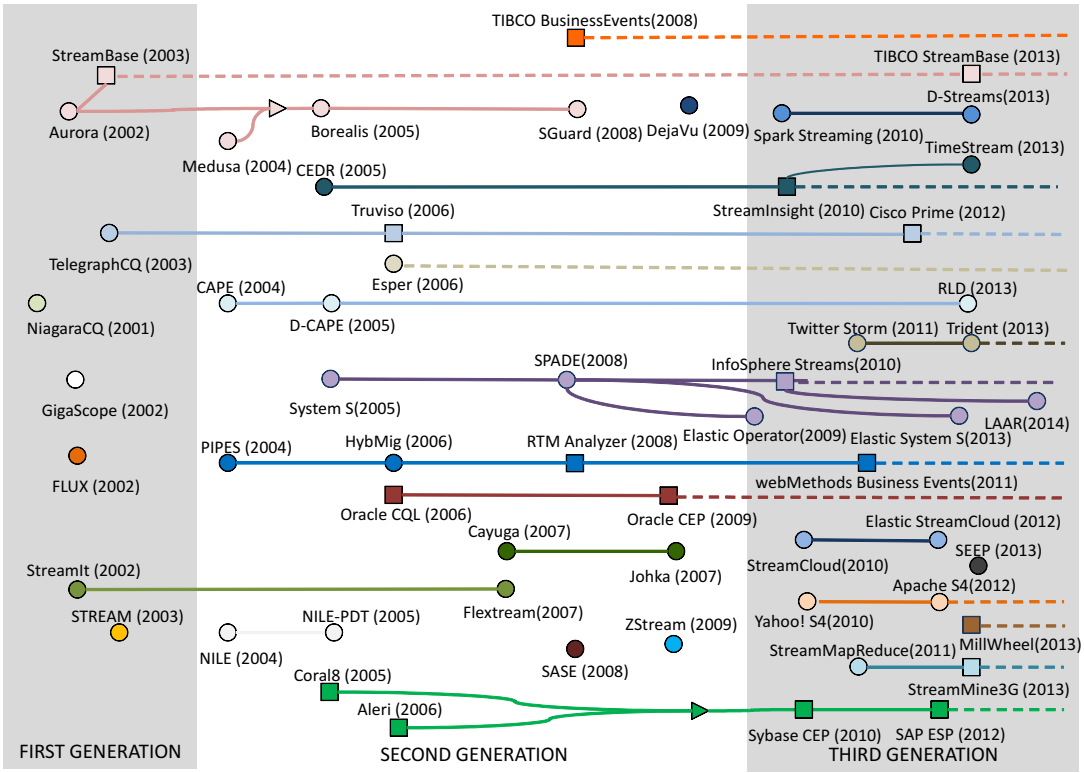


Figure 2.2: History of DSPSs [Hei+14a]

cremental query, which is executed periodically as a one-time SQL query, to find new matches to the original query as new data records are added to the database. However, such a "periodic re-execution" model for continuous queries cannot meet the *high throughput* and *low latency* requirements of modern stream-based applications. Hence, subsequently, specialized DSPSs were proposed. Figure 2.2, which is taken from the work by Heinze et al., summarizes the history of DSPSs. The first generation of DSPSs emerged in the early 2000s. Representatives include NiagaraCQ [Che+00], STREAM [BW01], Gigascope [Cra+02], Aurora [Aba+03], and TelegraphCQ [Cha+03]. Since then, a lot more endeavors were taken, aiming to provide higher system performance, richer operator set, and advanced features such as fault tolerance. These endeavors have led to the second generation of DSPSs including CEDR [Bar+07], Borealis [Ryv+06], SPADE [Ged+08], and Esper [Esp]. In recent years, the trend towards cloud computing has driven the development of large-scale, cloud-based DSPSs. Examples include StreamCloud [Gul+12], Apache S4 [Neu+10], Twitter Storm [Sto; Tos+14] and its recent descendent Twitter Heron [Kul+15], Spark Streaming [Zah+13], MillWheel [Aki+13], TimeStream [Qia+13], Stratosphere/Apache Flink [Ale+14; Fli], and Trill [Cha+14]. More details about this third generation of DSPSs can be found in [Hei+14a].

Despite the fact that a large number of DSPSs have been built and special endeavors [ABW06; KS09; Bot+10b] have been taken to clarify the semantics of executing continuous queries over data streams, to date, there are no established standards for data stream processing. This dissertation assumes a relational model [Cod70] for continuous queries, and adopts the semantic models introduced in the STREAM

project [ABW06]. The remainder of this section describes these semantic models. They are based on two data types—*stream*s and *time-varying relation*s—and three classes of query operators that operate over these two data types.

2.1.1 Data Model

Tuples

Under a relational model, a data item in a stream is referred to as a *tuple*. Formally, a tuple is a finite function that maps attribute names to values. The set of attributes, along with their associated domain, is called the *schema* of the tuple [Cod70].

Time

Time plays an important role in data stream processing. There are two notions of time: *application time* and *system time* [SW04a; Bot+10b; Aki+15]. Application time (also known as event time [Aki+15]) refers to the time at which a tuple is generated at a data source. It takes the clock time of the data source and conveys certain information about the application event that is represented by the tuple. For instance, a stock exchange may generate a tuple when a stock is traded; then the application time of this tuple conveys the time at which the corresponding stock-trading event has occurred. In contrast, system time refers to the time at which an event occurs in a DSPS and takes the clock time of the DSPS. For instance, the arrival of the above stock-trading tuple at a DSPS is a system event for the DSPS, and the arrival time of that tuple is a system time.

As in most prior work (e.g., [ABW06; Bot+10b; KS04; Jai+08]), this dissertation assumes a discrete, countably infinite time domain $\mathbb{T}$ with a total order. Each value from $\mathbb{T}$ is called a *time instant*. $\mathbb{T}$ can be represented, for instance, as non-negative integers $\{0, 1, \dots\}$, where 0 stands for the earliest time instant. Note that although both the application time and the system time can take values from this domain, they have different semantics.

Streams

Given a time domain $\mathbb{T}$ as defined above, a *stream* S is defined as a possibly infinite bag of tuples that conform to the schema of S. For a DSPS, a stream S_i that is received from an external data source is referred to as an *input stream* or a *base stream* [ABW06], where the subscript i is used to uniquely index an input stream in a DSPS. Let r_i denote the tuple arrival rate of the stream S_i; r_i may vary over time. Streams produced by query operators are referred to as *derived streams*. Particularly, a derived stream that contains the final results of a query is called a *result stream*. Let $e_{i,j}$ represent the j-th arrived tuple in the stream S_i. Let $e_{i.j}.ts \in \mathbb{T}$ represent the *application timestamp* of the tuple $e_{i,j}$; $e_{i.j}.ts$ indicates the generation time of $e_{i,j}$. Multiple tuples—either from the same stream or from different streams—could have the same application timestamp. Note that the application timestamps of stream tuples naturally define a temporal order among all tuples received by a DSPS.

In the remainder of this dissertation, the term *timestamp* is used exclusively to represent an application timestamp. In addition, the subscript of a tuple is omitted completely, or only the part that indicates the index of the input stream is kept,

wherever the omitted part of the subscript is not important in the context of the discussion.

Time-varying Relations

A *time-varying relation* R is a mapping from a time domain $\mathbb{T}$ to a finite but unbounded bag of tuples belonging to the schema of R. Compared with a relation in the standard relational model, a time-varying relation has the notion of time. Particularly, $R(\tau)$ represents the unordered bag of tuples contained in R at any time instant $\tau \in \mathbb{T}$, and is termed as an *instantaneous relation*.

In the remainder of this dissertation, the term *relation* is used to denote a time-varying relation and the term *static relation* is used to denote a relation in the standard relational model. Analogous to the classification for streams, relations that are input to a DSPS are called *input relations* or *base relations*, and relations that are produced by query operators are called *derived relations*.

Physical Representations of Streams and Time-varying Relations

Despite the semantic difference between streams and relations, both data types can share a common physical representation. For instance, in STREAM [ABW06] and Nile [Ham+04], both data types are represented as a sequence of *tagged tuples*. Specifically, a stream is represented as a sequence of timestamped *insertion* tuples, whereas a relation is represented as a sequence of timestamped *insertion* and *deletion* tuples to capture the evolving state of the relation. Instead of using deletion tuples, some other DSPSs (e.g., [Tuc+03; Li+05b; Kri+10]) use artificial tuples like *punctuations* to mark boundaries between instantaneous relations of a time-varying relation.

2.1.2 Query Model

Continuous queries are composed of query operators, whose semantics are defined in the query model. The query model adopted in this dissertation consists of three classes of query operators [ABW06]:

- *stream-to-relation* (S2R) operators, which produce one or more relations from one stream;
- *relation-to-relation* (R2R) operators, which produce one relation from one or more relations; and
- *relation-to-stream* (R2S) operators, which produce one stream from one relation.

Figure 2.3 depicts the relationships between these three classes of query operators. Note that the query model does not include *stream-to-stream* (S2S) operators. As claimed in [ABW06], the rationale for this decision is to "*exploit well-understood relational semantics (and by extension relational rewrites and execution strategies) to the extent possible*". Indeed, a stream-to-stream operator can be considered as a composition of a S2R operator, a R2R operator, and a R2S operator.

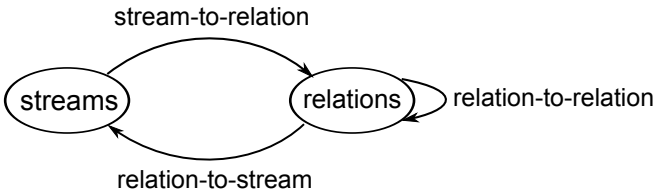


Figure 2.3: Query-operator classes and their relationships [ABW06]

Stream-to-Relation (S2R) Operators

A typical S2R operator is the *window operator*, which is also one of the most essential query operators in data stream processing [GÖ03a]. A window operator does not process the content of stream tuples, but is rather used to set bounds on an infinite stream to extract a finite set of tuples for further processing. Window operators are introduced into data stream processing mainly for the following reasons: (1) it is an approximation technique for solving the problem that computing over the entire history of a data stream would require unbounded amount of storage [Ara+02]; (2) In many real-world scenarios, the recent data is more important and relevant than the old data [AW04; GÖ03b]. Hence, rather than being viewed as an approximation technique, window operators, indeed, enable expressing the desired query semantics in those scenarios.

Depending on how a window operator sets the bounds over a stream and how the bounds move forward as the stream evolves, many different types of windows can be constructed [GÖ03a; PS06; ABW06]. A commonly-used window type in stream-based applications is the so-called *sliding window* [GÖ03b; AM04; Li+05a; Jin+10; Bha+14]. The two bounds of a sliding window, which are referred to as the *lower endpoint* EP_l and the *upper endpoint* EP_u, move simultaneously and at the same pace, so that the "distance" between the two bounds is fixed with respect to a certain pre-specified measuring unit. The distance between the two bounds of a window is called the *window size*, denoted by W. The measuring unit for specifying the window size can be the number of time units or the number of tuples falling into the window; and the resulting windows are referred to as the *time-based sliding window* and the *count-based sliding window*, respectively. The number of measuring units by which each time the window advances is called the *window slide*, denoted by β. The window slide can be specified in the number of time units or the number of tuples as well. A window with equal window slide and window size is called a *tumbling window* or a *jumping window*. In this dissertation, both the time-based window size and the time-based window slide use the notion of application time.

Example 2.1.1. Figure 2.4 shows an example of a time-based sliding window applied over a stream S_i. The window size W is three time units and the window slide β is one time unit. Each $w_{i,j}$ in the figure represents an instantaneous relation constructed by the window operator, capturing the state (i.e., the content) of the window at the moment that instantaneous relation is constructed. As new tuples of the stream S_i arrive, old tuples in the window are expired, and the expiration is determined based on the timestamps of tuples. For instance, at the arrival of the tuple $e_{i,5}$, the tuple $e_{i,1}$ is expired because $e_{i,1}.ts <= e_{i,5}.ts - W$; and the instantaneous relation $w_{i,2}$ is constructed. When the tuple $e_{i,6}$ arrives, all tuples with the timestamp 2 are

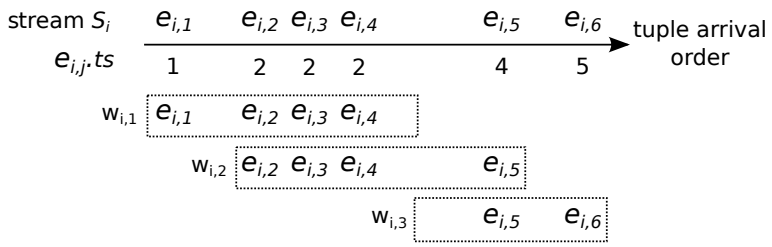


Figure 2.4: Example of time-based sliding window. ($W = 3$ time units, $\beta = 1$ time unit)

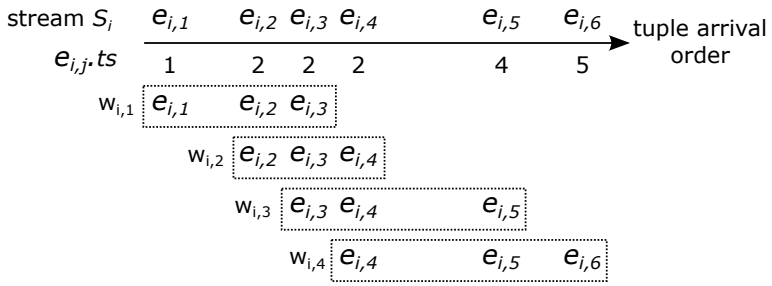


Figure 2.5: Example of count-based sliding window. ($W = 3$ tuples, $\beta = 1$ tuple)

expired and the instantaneous relation $w_{i,3}$ is constructed. It can be observed that, because the tuple arrival rate of the input stream may vary over time, instantaneous relations constructed by a time-based sliding window may contain different numbers of tuples.

Example 2.1.2. Figure 2.5 shows an example of a count-based sliding window for the same stream in Figure 2.4. The expiration of tuples is determined based on the number of tuples that are currently within the window. All instantaneous relations constructed by the window operator contain the same number of tuples.

Another representative window type is the *landmark window* [PS06]. A landmark window is defined by a *start predicate* and an optional *end predicate*. The window begins once an input tuple satisfying the start predicate is received. This tuple and all the following input tuples are added to the window until a tuple satisfying the end predicate—if specified—is received.

In the remainder of this dissertation, an instantaneous relation produced by a window operator is also referred to as an *instantaneous window*, or simply a *window* for short if there is no ambiguity.

Relation-to-Relation (R2R) Operators

Basically, each relational operator in a conventional DBMS has a R2R counterpart in the streaming context. This includes projection, selection (or filter), aggregate, join, and union. All these operators are widely used in stream-based applications [GÖ03a]. In addition to the above traditional operators, special operators such as map[1], pattern

[1] A map operator maps one input tuple into one or more different output tuples by applying a certain function on the input tuple. Example functions include format conversion, currency conversion, encryption, and any other scalar functions that can be found in a conventional DBMS.

matching, similarity searching, and frequent item mining are used very often in the streaming context as well, to serve the needs of applications that require advanced analysis of data streams. Generally, for a R2R operator with m ($m \geq 1$) input relations, each instantaneous relation produced in its output relation is the result of applying the operator logic over m instantaneous relations, one from each of the m input relations of the operator.

Relation-to-Stream (R2S) Operators

R2S operators are often used to convert the output of a R2R operator back to a stream. Three R2S operators were introduced in the STREAM system [ABW06]:

- *Istream*, which outputs only the newly inserted tuples in each instantaneous relation of the input relation.

$$Istream(R) = \bigcup_{\tau \geq 0} \left(R(\tau) - R(\tau - 1)\right), \text{ where } R(-1) = \emptyset$$

- *Rstream*, which outputs all tuples in each instantaneous relation of the input relation.

$$Rstream(R) = \bigcup_{\tau \geq 0} R(\tau)$$

- *Dstream*, which outputs only the newly deleted tuples in each instantaneous relation of the input relation.

$$Dstream(R) = \bigcup_{\tau > 0} \left(R(\tau - 1) - R(\tau)\right)$$

These R2S operators are sufficient in most application scenarios. Particularly, the *Istream* operator is used most often because its output stream reflects all new result tuples that are generated by a R2R operator over time. In the remainder of this dissertation, it is assumed that the *Istream* operator is used in all queries taken as examples and queries used in the evaluations.

System Operators

In addition to the three classes of logical operators described above, many DSPSs (e.g., Apache Storm [Sto], DataCell [LGI09], FUGU [Hei+14b], and MillWheel [Aki+13]) also have two special types of operators: *source* operators (e.g., *spouts* in Apache Storm, *receptors* in DataCell, and *injectors* in MillWheel) and *sink* operators (e.g., *emitter* in DataCell). Source and sink operators do not perform data-analysis tasks, but rather act as adapters for interacting with the external world of a DSPS. Specifically, a source operator brings external data into a DSPS, and a sink operator sends produced query results out of a DSPS.

In addition to the source and sink operators, a DSPS may also have operators for performing special tasks such as disorder handling (cf. Section 2.2) and load shedding (cf. Section 2.3.2). All these operators are referred to as *system operators*.

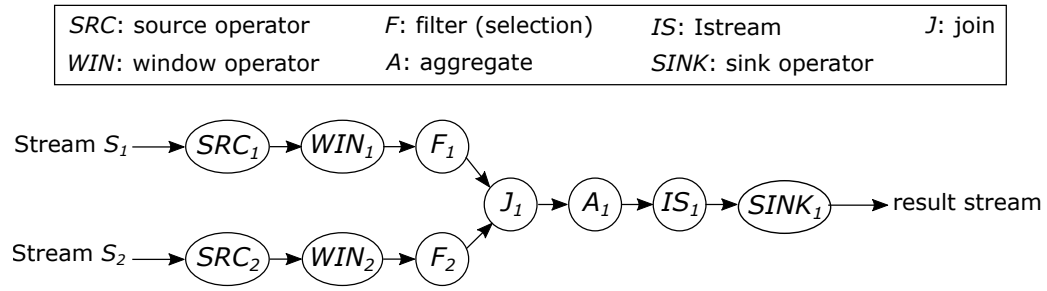


Figure 2.6: Example of logical query plan.

Continuous Queries

The processing logic of a continuous query can be expressed by a *logical query plan*, in the form of a directed acyclic graph (DAG) $G = (\mathcal{V}, \mathcal{ED})$. Each vertex $v_i \in \mathcal{V}$ in a logical query plan G represents a *logical query operator*. Each edge $ed_{ij} \in \mathcal{ED}$ represents the data flow from the operator v_i to the operator v_j. Using the terminology of DAG, the operator v_i is called a *parent* of v_j, and the operator v_j is called a *child* of v_i.

Normally, a continuous query first uses window operators to extract data that is most interesting to the application from the input streams of the query. Following the window operators, a graph of R2R operators are used to perform the actual data-analysis task. R2R operators that follow window operators are also called *window-based operators*. Then, a R2S operator is used to transform the relation produced by the graph of R2R operators back to a stream, which can either be sent back to the application as a result stream, or be consumed by another graph of window and R2R operators for further processing. An example of a logical query plan is shown in Figure 2.6. Like a conventional SQL query, a continuous query may have multiple semantically-equivalent logical plans.

As Arasu et al. [ABW06] have discussed, one major drawback of not having S2S operators in the query model is that even a simple data-filtering task performed over a stream would require a count-based tumbling window of one-tuple size, one relational selection operator, and one *Istream* operator. However, logical operators are used only to express the processing logic of a continuous query, and the actual data processing in a DSPS is performed by *physical operator*s over physically represented streams and relations. A DSPS can map a chain of logical operators following a certain pattern to a more efficient physical implementation. For instance, for an operator chain in a logical query plan that has a similar pattern as the filtering task described above, a DSPS can map it to a single, *composite* physical operator. Basically, for any R2R operator in a logical query plan, if all its parent operators are window operators and its child operator is a R2S operator, then the parent window operators, the R2R operator itself, and its child R2S operator can all be mapped to a single, composite physical operator. The DAG formed after translating all operators in a logical query plan to physical operators is called a *query execution plan*.

Operator Implementation

A logical query operator can be implemented in many different ways. A basic requirement is to guarantee that the semantics of a physical operator is consistent

with the semantics of its corresponding logical operator. Modern stream-based applications require processing data streams with high throughput and low latency. Hence, an operator implementation must also be efficient to meet the performance requirements of applications.

A commonly-used operator-implementation technique to support efficient data stream processing is the so-called *incremental computation* [RR93; Gha+07]. Specifically, a query operator is said to be *incrementally computable* if changes in the output of the operator can be determined based on only changes in the operator's input(s) and certain internally-maintained operator states; otherwise, the query operator is not incrementally computable. The specific operator states that need to be maintained depend on the operator logic.

Many window-based operators are incrementally computable, including window-based select, window-based join, and window-based aggregate that computes functions like COUNT, SUM, and AVG. Take a COUNT aggregate as an example. Assume that initially the input relation of the operator contains three tuples, then the initial COUNT result is three. To support incremental computation, the operator maintains the current COUNT result as an internal state. Now assume that two more tuples are added to the input relation of the operator. To get the new COUNT result, the operator can simply increase the internally-maintained result by two and output the updated result, which is five. In contrast to a COUNT aggregate, to compute exact results, a MEDIAN or a QUANTILE aggregate needs to re-scan the current content of the input relation if the input relation has changed; hence, they are not incrementally computable.

2.1.3 Query Execution Model

To adapt to the "push" characteristic of data streams, existing DSPSs often adopt the *pipelined query execution model* [Gra93]. With pipelined execution, physical query operators are connected via a buffering mechanism like queues. Any two neighboring operators can be seen as a producer-consumer pair, and can run in parallel, embodying the so-called *pipelined parallelism* [HM94]. Pipelined query execution allows a DSPS to exploit the power of modern multiprocessor machines, as well as the increasingly-prevalent distributed processing environment, to achieve a higher processing performance.

2.2 Handling Data Imperfection in Data Streams

In real-world scenarios, the streams input to a DSPS are often imperfect [GÖ03a; Sto+05; KL09]. There are different types of data imperfection. Certain types of data imperfection can be remedied by a DSPS, whereas others cannot because of the lack of the "ground truth". For the remediable data imperfection, on the one hand, executing continuous queries over streams without handling the data imperfection may lead to unexpected query results, and therefore degrade the quality of the produced query results; on the other hand, handling the data imperfection often introduces additional overhead and therefore degrades the processing performance. Hence, handling remediable data imperfection in data streams involves a tradeoff between the performance and the query-result quality. Moreover, this tradeoff is inevitable,

because a DSPS can control neither how data is generated by external data sources, nor how the generated data is transferred to the system. This section discusses common types of data imperfection in streams, their relevance to the performance versus result-quality tradeoff (Section 2.2.1), and existing work on handling these types of data imperfection (Section 2.2.2).

2.2.1 Common Types of Data Imperfection

Data Uncertainty

One typical type of data imperfection in data streams is the *data uncertainty*. Uncertain data is very common in streams that are generated by sensors, RFID readers, or GPS devices; because these data sources are sensitive to the orientation of reading and environmental factors such as interference. Certain conversion actions performed at a data source often introduces data uncertainty as well, e.g., the conversion of voltage measurements into Celsius at temperature sensors, or the analog-to-digital conversions [KBS06; JGF06; KD08]. Data uncertainty can be interpreted and measured in different aspects. Aspects that are considered widely in literature include *confidence* and *completeness* [CG07; KL09; Tra+12; GL12].

Confidence describes the *belief*, or likelihood, of a given data item. It is often quantified in the form of *statistical probabilities*. Confidence can be assigned either at the tuple level, describing the likelihood that a given tuple appears, or at the attribute level, describing the likelihood that a tuple attribute takes a certain value. The former is called the *tuple-level uncertainty*, and the latter is called the *attribute-level uncertainty* [Sar+09]. Under the probability theory, the overall uncertainty of a tuple or an attribute can be represented by a random variable, which is associated with a probability distribution over the possible values of the variable. The domain of the variable could be discrete or continuous. The probabilities of all possible values sum up to one, meaning that the variable will take one of those values with certainty.

Continuous queries executed over streams with tuple-level or attribute-level uncertainty produce uncertain query results. Namely, each result tuple is associated with probabilities as well, describing the tuple-level or the attribute-level uncertainties of the result tuple. Streams with this kind of data uncertainty is not remediable, and one cannot infer what the *exact* query results are. As a result, there is no reference for measuring the quality of the produced query results. For this reason, tuple-level and attribute-level uncertainties do not really cause a performance versus result-quality tradeoff.

Completeness addresses the issue of missing tuples in a data stream, which often results from unreliable transmission protocols, failures in data sources, or failures in the transmission infrastructure. Without a-priori knowledge about how the data of an input stream is generated (e.g., the measurement frequency of the sensor that generates the data), a DSPS often even cannot recognize the fact that there are missing tuples in the input streams. Even if such a-priori knowledge is present, the DSPS only can estimate the values of the missing tuples [Gru+10b; Gru+10a; Gao+16], but cannot infer the exact values. As a result, the DSPS cannot infer the exact results of a query executed over streams with missing tuples either. Hence, similar to the case of tuple-level and attribute-level uncertainties, the incompleteness within the input streams of a DSPS does not really cause a performance versus result-quality tradeoff.

The loss of tuples could happen within a DSPS as well, especially in distributed DSPSs. This type of tuple loss is remediable by the DSPS, by using certain fault-tolerance mechanisms. Many existing DSPSs, e.g., Storm Trident [Tri], Spark Streaming [Zah+13], MillWheel [Aki+13], and Apache Flink [Fli], provide an *at-least-once*, or even an *exactly-once*, data-processing guarantee so that eventually no tuple is lost in any derived relation or stream within the DSPS. The basic idea for achieving such data-processing guarantees is to use an acknowledgment mechanism to detect tuples that are lost during the processing, and re-deliver those lost tuples. However, re-delivering tuples often introduces disorder into data streams, which is another typical type of data imperfection.

Stream Disorder

Recall that stream tuples are often tagged with timestamps when they are generated at data sources, and the timestamps naturally define a temporal order among all tuples (cf. Section 2.1.1). In general terms, *stream disorder* refers to the situation that the order in which stream tuples arrive, at either source operators of a DSPS or any data-analysis operators, is different from the order in which they were generated at external data sources or the parent operators of the operator that receives the tuples. Stream disorder enters in two forms: First, an individual stream could be *out of order*; namely, tuples within the stream do not arrive in non-decreasing timestamp order. This form of stream disorder is called the *intra-stream* disorder. Second, for operators with multiple input streams, even if each individual input stream is timestamp-ordered, the tuples from different input streams could arrive out of order. This form of disorder is called the *inter-stream* disorder.

Stream disorder is ubiquitous in real-world streams transmitted across a network because of the inherent network asynchrony. For instance, in a sensor network, tuples sent from different sensors to the gateway node may experience different delays, and arrive at the gateway node in an arbitrary order. Even tuples from the same sensor may arrive at the gateway node out of order, if an unreliable transmission protocol (e.g., UDP) is used. Another typical cause of stream disorder is the MapReduce-style data-parallel processing, where data is partitioned and processed by several parallel instances of a query operator [DG08]. Results from the parallel operator instances may arrive out of order at the result merger because of the various processing speeds of the instances [Hir+14]. In addition, certain implementations of query operators like join and union may produce out-of-order results as well [TM11].

Formally, let ${}^iT$ represent the *local current time* of a stream S_i, which is defined as the maximum timestamp among the so-far-observed tuples in S_i, i.e., ${}^iT = max\{e_{i,j}.ts \mid e_{i,j} \in S_i\}$. The stream S_i is considered to have *intra-stream disorder* if it contains tuples $e_{i,j}$ and $e_{i,k}$ such that $j < k$ and $e_{i,j}.ts > e_{i,k}.ts$. The tuple $e_{i,k}$ in this case is called an *out-of-order tuple*, or a *late arrival*, in S_i. Moreover, for a tuple e_i in any stream S_i, the *delay* of the tuple, denoted by $delay(e_i)$, is defined as the difference between the value of ${}^iT$ updated at the arrival of e_i and the timestamp of e_i itself, i.e., $delay(e_i) = {}^iT - e_i.ts$.

In this dissertation, it is assumed that for any continuous query deployed in a DSPS, the clocks of the external data sources involved in the query are synchronized. Existing clock synchronization techniques such as VHT [SDS10], FTSP [Mar+04], as well as the work proposed in [FC97] can provide high-resolution and high-accu-

racy synchronization. Based on this assumption, the inter-stream disorder can be described by the *time skew* between a pair of streams S_i and S_j ($i \neq j$), denoted by $skew(S_i, S_j)$. Specifically, $skew(S_i, S_j)$ is defined as the absolute difference between the local current time of S_i and the local current time of S_j, i.e., $skew(S_i, S_j) = |{}^iT - {}^jT|$. As ${}^iT$ and ${}^jT$ are updated by newly-arrived tuples in S_i and S_j respectively, $skew(S_i, S_j)$ often varies during the lifetime of S_i and S_j. Given m streams, the stream with the smallest current local time T is referred to as the *slowest* stream in terms of the timestamp progress.

Stream disorder influences the results of queries containing operators that are sensitive to the timestamp order of the input tuples. Many window-based operators (cf. Section 2.1.1) are order-sensitive operators. The reason is that stream disorder makes it difficult for a window operator to decide when to produce instantaneous windows to be processed further by the R2R operator that follows the window operator. If an instantaneous window is produced before all tuples that fall into the scope of this instantaneous window have arrived, then the produced instantaneous window is indeed *incomplete*, and applying the R2R operator on it may produce unexpected results. The exact results of a query executed over disordered streams can be inferred: they are the query results that would be produced if the input streams do not have disorder. Stream disorder is remediable within a DSPS, at the cost of degraded processing performance. Hence, stream disorder causes an inevitable tradeoff between the performance and the query-result quality.

Because data uncertainty does not really cause a performance versus result-quality tradeoff, and the tuple-loss problem within a DSPS eventually transforms to the stream-disorder problem if the DSPS provides data-processing guarantees via fault-tolerance mechanisms, this dissertation takes stream disorder as an representative case of data imperfection, and studies the performance versus result-quality tradeoff caused by stream disorder.

2.2.2 Approaches for Handling Data Imperfection

This section briefly discusses the state-of-the-art approaches for handling the two common types of data imperfection described in the previous section.

Data Uncertainty

Handling data uncertainty within streams have attracted a lot of research interest in the past years. A recent survey can be found in [AY09]. The majority of the existing work (e.g., [Jay+07; CG07; Hua+08; ZLY08; Jin+10]) focuses on tuple-level and attribute-level uncertainties, and adopts the *possible worlds* semantics, which was first introduced for *probabilistic databases* [DS07]. In brief, an uncertain stream with tuple-level or attribute-level uncertainties has many, normally exponentially-large, possible instances. Each instance is constructed from a valid combination of tuples in the stream, and is termed a *possible world*. Hence, the uncertain stream can be viewed as defining a probability distribution over all the possible worlds.

Example 2.2.1. Consider a simple stream $S = (\langle a, \frac{1}{2}\rangle, \langle a, \frac{1}{3}\rangle, \langle b, \frac{1}{4}\rangle)$ with tuple-level uncertainty. Take the first tuple $\langle a, \frac{1}{2}\rangle$ as an example; a is the value of the only

attribute of the tuple, and $\frac{1}{2}$ is the tuple-level confidence. The timestamps of the tuples are omitted for simplicity. This uncertain stream has eight possible worlds as shown in the table below.

Possible World	(a)	(a)	(b)	(a,a)	(a,b)	(a,b)	(a,a,b)	$\emptyset$
Probability	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{12}$	$\frac{1}{8}$	$\frac{1}{12}$	$\frac{1}{24}$	$\frac{1}{24}$	$\frac{1}{4}$

A query executed over an uncertain stream under the possible world semantics essentially defines a probability distribution over the space of query results for all possible worlds. Different types of queries have been studied in prior work. Just to name a few, Jayram et al. [Jay+07] studied commonly-used basic aggregate queries including SUM, COUNT, AVERAGE, and MEDIAN. Cormode and Garofalakis [CG07] studied more complex aggregate queries, including the number of distinct values and join sizes. Zhang et al. [ZLY08] studied *frequent-item* queries. The work of [Hua+08] and [Jin+10] studied *top-k* queries; particularly, the latter studied top-k queries with sliding windows. Because the probability distribution defined by a query has a potentially-enormous size and complexity, some work, e.g., [CG07], focused on characterizing such a distribution through its key moments such as *expectation* and *variance*.

The possible world semantics is not applicable for uncertain data that is naturally modeled using continuous random variables, because the possible values of a continuous random variable are infinite and cannot be enumerated. To address this problem, Tran et al. [Tra+12] proposed the CLARO probabilistic DSPS, which supports executing relational queries. The foundation of CLARO is a *mixed-type* data model, which captures the tuple-existence uncertainty and uses Gaussian mixture distributions to characterize continuously-valued uncertain attributes.

Stream Disorder

A variety of disorder handling approaches have been proposed, especially for handling the intra-stream disorder. Based on the underlying mechanism, these approaches can be grouped into four categories: buffer-based approaches [Aba+03; BSW04; MP13a], punctuation-based approaches [Li+05b; Li+08; Liu+09; SW04a], speculation-based approaches [Bar+07; Bri+08], and hybrid approaches [Kri+10; MP13b]. Buffer-based approaches use buffers to reorder tuples within an individual stream to handle the intra-stream disorder, and also use buffers to synchronize different streams to handle the inter-stream disorder. Punctuation-based approaches rely on special tuples embedded in streams to communicate the stream progress. Speculation-based approaches produce query results speculatively, and apply a compensation technique to correct early-emitted results when out-of-order tuples are observed. Hybrid approaches combine two of the former three approaches. A more detailed discussion of these disorder handling approaches will be provided in Section 3.5.

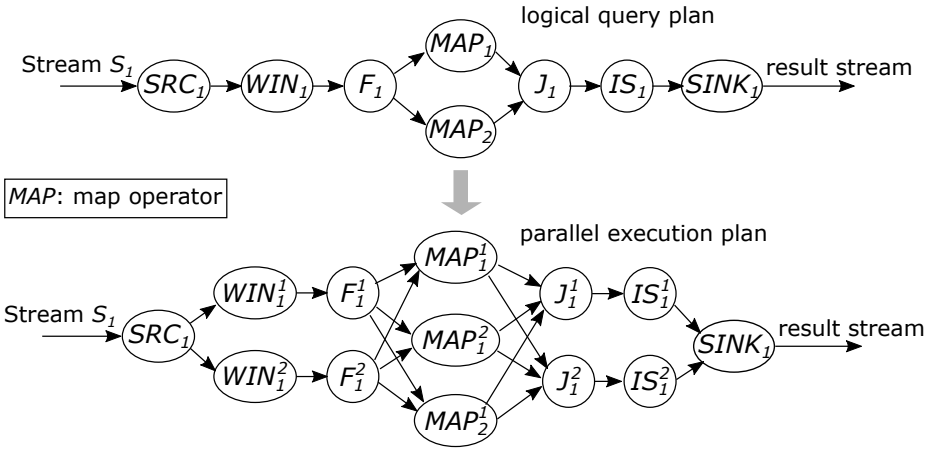


Figure 2.7: Parallelizing query-operators for improving processing performance.

2.3 Handling System Limitations of DSPSs

Tradeoffs between the performance and the query-result quality in data stream processing can also be caused by resource or implementation limitations of a DSPS itself. In contrast to the tradeoffs caused by data imperfection, tradeoffs caused by system limitations can be reduced, even eliminated, by enhancing the DSPS itself. In the following, Section 2.3.1 presents different categories of system-enhancement approaches; Section 2.3.2 discusses typical techniques for trading query-result quality for performance when those system-enhancement approaches failed to eliminate the tradeoff for certain reasons (e.g., there may be limited budget for adding enough resources to the system).

2.3.1 Approaches for Enhancing a DSPS

A DSPS can be enhanced in many different ways. This section presents five major categories of system-enhancement approaches. They are adding more computation resources, using smarter operator implementations, employing columnar processing, exploiting hardware acceleration, and combining different technologies.

Adding more computation resources is a natural solution for enhancing a DSPS. It is also one of the key design principles of modern cloud-based DSPSs such as Twitter Heron [Kul+15] (the descendant of Twitter Storm [Tos+14]), Spark Streaming [Zah+13], Apache Flink [Fli], and MillWheel [Aki+13]. The performance improvement comes mainly from distributed processing and operator parallelization. More specifically, as shown in Figure 2.7, an operator in a logical query plan could have several parallel physical operator instances. Each parallel physical instance of a logical operator processes a partition of the logical operator's original input. For the example in Figure 2.7, each of the operators WIN_1, F_1, MAP_1, J_1, and IS_1 has two parallel physical instances, and each of the other logical operators has only one physical instance.

In principle, each physical operator instance can run exclusively on a processing node. When the workload of a DSPS increases and cannot be handled by the currently-running processing nodes, more processing nodes can be added to the system. The

operator instances running on a processing node that gets overloaded can then be migrated to and distributed over those newly-added nodes. Moreover, an operator instance that caused the overload can be parallelized further, by further partitioning the input of the operator to be processed by more parallel physical instances. When applying such a parallel and distributed execution for a query, it is important to guarantee that the semantics of the query is not changed.

One major challenge for distributed and parallel DSPSs is to determine when and how to scale the system to deal with overload situations automatically, and at the same time to utilize system resources efficiently with respect to the monetary cost [Hei+14a]. This behavior is also known as *elasticity*, which was first introduced in System S [Sch+09]. Cost-efficiency is especially important for could-based DSPSs, because they adopt a "pay per use" model. Users of a cloud-based DSPS desire a high ratio between the monetary cost spent and the *quality of service* (QoS) provided by the system. A lot of research has been done in this area, including the recent work of Elseidy et al. [Els+14], which focuses on join queries, and the work of Gedik et al. [Ged+14] for general-purpose stream-based applications.

The second natural solution for enhancing a DSPS is to **use better operator implementations**. A great deal of work has been done along this line. For example, the *just-in-time* method proposed in [YP08] allows a consumer operator to send its feedback on the demand for input to its producer operator; and the producer operator can then selectively generate results based on this feedback. This method allows saving both the CPU time and the memory consumption. Slider [Bha+14] and Reactive Aggregator [Tan+15] are two systems focusing on executing sliding-window operations efficiently by using incremental computation. Particularly, Reactive Aggregator is general and can handle non-invertible and non-commutative aggregates, as well as non-FIFO windows. BiStream [Lin+15] is another example in this area. It is a scalable distributed stream join system, which employs a join-biclique model instead of a join-matrix [SY93] model. With the join-biclique model, processing nodes of the system are organized as a complete bipartite graph. This model is more memory-efficient and communication-efficient than the join-matrix model, because each input partition needs to be stored by only one processing node.

In recent years, it was shown that column-oriented processing has higher performance than row-oriented processing for analytical queries [AMH08]. Therefore, **employing column-oriented processing** is another way to enhance a DSPS. A representative work along this line is Trill [Cha+14]. Like Spark Streaming [Zah+13], Trill exploits batching for high throughput. However, different from Spark Steaming, Trill uses a columnar data layout within batches, and generates operator source codes on the fly to compute over the columnar batches.

Despite software-based approaches, **exploiting hardware acceleration** is another option for system enhancement. The idea is to design stream-processing algorithms that can fully exploit the high degree of parallelism within modern hardware like multi-core processors, GPU, and FPGA. Representative work in this area includes the handshake-join algorithm [RTG14; TM11] designed for multi-core processors. With the handshake-join, the two streams being joined flow by each other in opposite directions, generating join results as the streams pass by. This algorithm can be parallelized easily over all available cores, with each core processing one segment of the window over which the join operation is applied. The HELLS-join algorithm [Kar+13]

utilizes the high memory-bandwidth of GPU for parallel tuple comparison. FPGA is another type of hardware which has high parallelism inherently. Existing work has studied how to use FPGA to accelerate join queries [RTG14], frequent-item queries [TMA10], and even general select-project-join (SPJ) queries [NSJ13].

In addition to the four categories of system-enhancement approaches described above, there is one category of system-enhancement approaches that follow the philosophy of "no one size fits all", and **leverage advantages of different technologies**. For instance, the Cyclops platform [LHB13; LB13] federates Esper [Esp], Storm [Sto], and Hadoop [Apa] for executing window-based aggregate queries. It picks the most suitable system for a query based on properties such as the size and the slide of the window operator in the query, and the applied aggregate function. To support federating different DSPSs, Duller et al. [Dul+11] proposed a middleware platform called ExoP. ExoP virtualizes components of a DSPS and provides well-defined, extensible interfaces for exchanging data between different DSPSs.

One major challenge for such hybrid systems is to determine the optimal execution plan for a given query, which is known as the *query optimization* problem [Sel+79]. The optimization could be done at the query level by choosing the most suitable DSPS for the entire query, or at the operator level by determining the most suitable DSPS for each operator in the query. Compared with the query-level optimization, the operator-level optimization often can produce execution plans with higher performance. However, the search space of possible execution plans in operator-level optimization is also much larger than that in query-level optimization. In addition, different from executions plans of traditional SQL queries, execution plans of continuous queries have a unique property called *feasibility*, which defines the ability of an execution plan to keep up with the tuple arrival rates [AN04]. The feasibility property should be taken into account when optimizing continuous queries.

2.3.2 Approaches of Trading Result-Quality for Performance

In real-world scenarios, it may happen that the system-enhancement approaches described in the previous section still cannot support a DSPS to produce exact query results under a given workload. For instance, a DSPS may be unable to get enough computation resources because of a limited monetary budget. In this case, to avoid system overload, the DSPS can trade the quality of query results for performance for applications that do not require perfect query results. For example, network-monitoring and weather-monitoring applications normally can tolerate imperfect query results.

There are two commonly-used techniques for trading query-result quality for performance. The first technique is **load shedding** [Tat+03], which is used to deal with sudden spikes in the tuple arrival rate. The basic idea is to drop a fraction of input tuples and produce approximate query results based on the remaining tuples. Here, the concerned system resource is mainly the CPU. The major challenge concerning the performance versus result-quality tradeoff is to determine how to shed the load to provide certain user-desired performance guarantees, e.g., a guarantee on the end-to-end latency, and at the same time to maximize the quality of the produced query results. Prior work has addressed this challenge for different types of queries. For instance, Gedik et al. [Ged+07] studied how to maximize the output rate of MSWJ

queries under load shedding; Mozafari et al. [MZ10] studied how to minimize the error in the results of window-based aggregate and mining queries.

The second typical technique for trading query-result quality for performance is **approximate query processing (AQP)** based on the usage of data synopses such as samples, sketches, and histograms [AY07; Cor+12]. AQP is often applied when, in general, every query result has to be computed based on a large amount of tuples. In this case, it is neither time-efficient nor space-efficient to produce exact query results based on the original input. AQP is normally applied to answer aggregate queries, and can provide certain (probabilistic) accuracy guarantees for the produced query results. The accuracy guarantees are related to certain parameters of the used synopsis, e.g., the size of the data sample or the number of buckets in a histogram. Indeed, these parameters determine the degree of the tradeoffs between the space consumption, the processing time, and result accuracy. A survey of AQP can be found in [Cor+12].

2.4 Summary

This chapter first discussed basic concepts in data stream processing and described the query-processing semantics adopted in this dissertation, which set up the conceptual context of this dissertation. Afterwards, focusing on the topic of the tradeoffs between the performance and query-result quality, it looked into the two root causes of the tradeoffs—data imperfection and system limitation. For the aspect of data imperfection, two common types of data imperfection—data uncertainty and stream disorder—along with the state-of-the-art approaches for handling them were discussed. For the aspect of system limitation, five categories of system-enhancement approaches were identified, and approaches for trading query-result quality for performance in case those system-enhancement approaches failed were discussed.

3

Providing Flexible Tradeoff via Quality-Driven Disorder Handling

Section 2.2 discussed common types of data imperfection and their relevance to the tradeoffs between the performance and the query-result quality. Based on that discussion, this and the next two chapters focus on the data imperfection—stream disorder—to study the research question "*how to provide flexible and user-configurable tradeoffs*". Particularly, in this chapter, Section 3.1 elaborates the performance versus result-quality tradeoff caused by stream disorder, and motivates the idea of *buffer-based, quality-driven disorder handling* (QDDH); Section 3.2 introduces the background knowledge of buffer-based disorder handling; Section 3.3 provides an overview of the generic buffer-based QDDH framework proposed in this dissertation; Section 3.4 drills down to the methods for performing quality-driven buffer-size adaptation; and finally Section 3.5 discusses existing disorder handling approaches in detail, as well as other work that is related to disorder handling.

3.1 Motivation

It was mentioned in Section 2.2.1 that stream disorder may impair the result quality of queries that involve order-sensitive operators such as window-based operators. This claim is now illustrated with the following two examples.

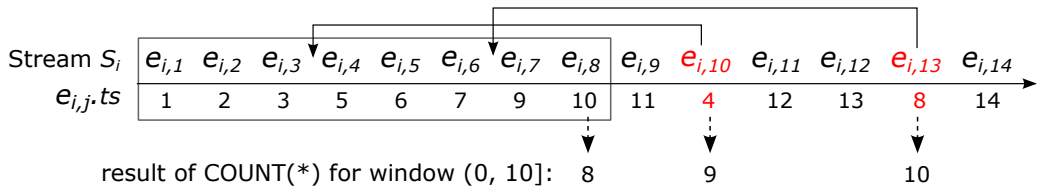


Figure 3.1: Effect of stream disorder on the result quality of sliding-window aggregate queries.

Example 3.1.1. Figure 3.1 demonstrates the effect of disorder handling on the results of sliding-window aggregate (SWA) queries. Recall from Section 2.1.1 that $e_{i,j}$ represents the j-th arrived tuple in the stream S_i. In the stream S_i in Figure 3.1, the two

tuples $e_{i,10}$ and $e_{i,13}$, whose timestamps are 4 and 8 respectively, are two out-of-order tuples; because for each of them, tuples with larger timestamps have arrived earlier. Assume that a sliding-window COUNT(*) query is executed over this stream and the window size is 10 time units. Logically, the instantaneous window with scope $(0, 10]$ should be constructed at the arrival of the tuple $e_{i,8}$, whose timestamp is 10. The result of COUNT(*) for this instantaneous window computed at this moment would be 8. However, this result is inaccurate; because the two out-of-order tuples, $e_{i,10}$ and $e_{i,13}$, fall into the scope $(0, 10]$ as well, but are not counted. If the COUNT(*) result for this instantaneous window is not computed until the arrival of the tuple $e_{i,10}$, then a more accurate result, 9, can be produced. To obtain the exact query result, which is 10, the computation of this result must be delayed until the arrival of the tuple $e_{i,13}$.

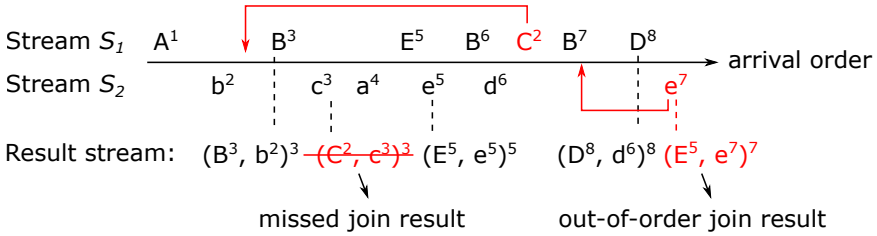


Figure 3.2: Effect of stream disorder on the result quality of sliding-window join queries. The size of the window on each stream is 3 time units.

Example 3.1.2. Figure 3.2 demonstrates the effect of stream disorder on the result quality of m-way sliding-window join (MSWJ) queries. For simplicity, a 2-way equi-join $S_1 \bowtie S_2$ is considered. A sliding window of 3 time units is applied on each input stream of the join. In general, an MSWJ query works as follows [GÖ03b; VNB03; WR09]: for each tuple e arrived from any input stream, the expired tuples in the windows on all the other input streams are detected and invalidated based on the timestamp of e. The tuple e is then joined with all tuples remaining in those windows, and the result tuples, which satisfy the predefined *join condition* $p^{\bowtie}$, are produced. The timestamp assigned to a result tuple is the maximum timestamp among its deriving input tuples.

In Figure 3.2, an input tuple is represented by x^{ts}, where x is the value of the tuple's join attribute, and ts is the tuple's timestamp. To differentiate tuples of the stream S_1 from tuples of the stream S_2, capital letters are used to represent attribute values of S_1 tuples. The timestamp of a result tuple is denoted by superscript as well. In this example, the tuple C^2 is an out-of-order tuple in the stream S_1. Without handling the stream disorder, the result tuple $(C^2, c^3)^3$, which can be derived from the tuple C^2 of S_1 and the tuple c^3 of S_2, would be missed. The reason is that, given the window size on each input stream being 3 time units, by the time the tuple C^2 arrives, the matching tuple of C^2 in the stream S_2, which is c^3, has expired from the current window on S_2 because of the arrival of the tuple B^6. The stream S_2 is timestamp-ordered; however, the tuple e^7 of S_2 arrives after the tuple D^8 of S_1, and hence is out of order from the perspective of the join operator. In this case,

although the result tuple $(E^5, e^7)^7$ can still be produced[1], it is out of order in the result stream. Out-of-order result stream is unacceptable in many scenarios, e.g., when the output is consumed for feedback control. Sorting the produced result tuples can avoid out-of-order result tuples [HAE05; RTG14]; and sorting the input tuples to process them in a timestamp order can avoid both missed and out-of-order result tuples [SW04a; HAE05]. However, in either case, the end-to-end latency would be increased. Performing incomplete sorting can reduce the latency incurred by disorder handling, but a fraction of the true join results would get lost[2].

Example 3.1.1 and Example 3.1.2 show that the main performance metric influenced by disorder handling is the *end-to-end latency*, and there is an inevitable tradeoff between the end-to-end latency and the produced query-result quality when doing disorder handling. Specifically, the query-result quality refers to the accuracy of each produced aggregate result for SWA queries, and refers to the completeness of the produced result set for MSWJ queries.

Many stream-based applications, e.g., network-traffic monitoring and environment monitoring, do not demand perfect query results. To support timely data analysis, these applications accept trading query-result quality for low latency. However, query results with significantly-low quality may cause severe consequences if further actions are taken based on these results. Hence, it is still desired that the query-result quality is controlled at an acceptable level. Moreover, different applications often prefer different points in the spectrum of the tradeoff between the latency and the query-result quality.

Based on the above observations, in this dissertation, it is argued that a disorder handling approach should support a user-configurable tradeoff between the latency and the query-result quality. Existing disorder handling approaches either do not provide this configurability (e.g., [BSW04; SW04a; Liu+09; CGM10; MP13a]), or support disorder handling only under user-specified latency constraints (e.g., [Aba+03; Li+08]). As a complement to the state of the art, this dissertation proposes the concept of *quality-driven disorder handling (QDDH)*. The objective is to *minimize the time spent on waiting for out-of-order tuples, thus the end-to-end latency incurred by disorder handling, while honoring user-specified requirements on the query-result quality*. From Example 3.1.1 and Example 3.1.2, it can be observed that the result-quality metric for a continuous query depends on the type of the query. Because aggregate queries and join queries with time-based sliding-windows are at the heart of many stream-based applications, this dissertation focuses on studying how to apply QDDH for these two types of queries.

Recall from Section 2.2.2 that there are three categories of basic disorder handling approaches, namely, buffer-based approaches, punctuation-based approaches, and speculation-based approaches. In this dissertation, the concept of QDDH is implemented on top of buffer-based disorder handling approaches, which will be described in the next section in detail. However, the QDDH concept can be implemented

[1]Note that some join algorithms (e.g., [KNV03; SW04b]) would miss the result tuple $(E^5, e^7)^7$; because upon receiving a new tuple, these algorithms invalidate expired tuples in the windows on all input streams. Hence, at the arrival of the tuple D^8, the tuple E^5 would expire from the window on S_1.

[2]Incomplete sorting of result tuples leads to the loss of join results as well, because result tuples that are still out of order after sorting are discarded to fulfill the "in-order output" requirement.

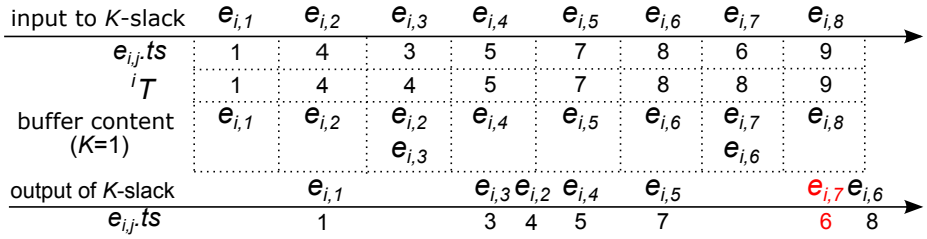


Figure 3.3: Example of using K-slack to handle the intra-stream disorder.

in combination with punctuation-based and speculation-based disorder handling approaches as well, which will be discussed in Section 3.5.

3.2 Buffer-Based Disorder Handling

3.2.1 Handling Intra-Stream Disorder

A well-known buffer-based approach for handling the intra-stream disorder is the K-slack algorithm [BSW04; Li+07; MP13a], where the configurable parameter K represents the buffer size. The basic idea of K-slack is as follows: To handle the disorder within a stream S_i, tuples from the stream are first inserted into a buffer of K_i time units. Within the buffer, tuples are sorted based on their timestamps. The local current time ${}^iT$ of the stream S_i (cf. Section 2.2.1) is used to help determining when to release tuples from the K-slack buffer. Recall that ${}^iT$ tracks the maximum timestamp among all S_i tuples that have arrived so far. Each time ${}^iT$ is updated by a new tuple from S_i, each tuple $e_{i,j}$ in the K-slack buffer, whose timestamp satisfies the condition in Eq. (3.1), is released from the buffer. All tuples satisfying the condition are released in the timestamp order.

$$e_{i,j}.ts + K_i \leq {}^iT \tag{3.1}$$

Example 3.2.1. Figure 3.3 gives an example of the intra-stream disorder handling using a K-slack buffer, where the buffer size K is 1 time unit. When the tuple $e_{i,1}$ arrives, the local current time ${}^iT$ of the stream is updated to 1 (i.e., ${}^iT = e_{i,1}.ts = 1$), and $e_{i,1}$ is inserted into the buffer. The tuple $e_{i,1}$ cannot be released at this moment because it does not satisfy the release condition defined in Eq. (3.1). When the tuple $e_{i,2}$ arrives, ${}^iT$ is updated to 4. Now, the tuple $e_{i,1}$ satisfies the release condition and is emitted from the buffer. The tuple $e_{i,3}$ is an out-of-order tuple. According to the definition of the delay of a tuple (cf. Section 2.2.1), the delay of $e_{i,3}$ is $delay(e_{i,3}) = {}^iT - e_{i,3}.ts = 4 - 3 = 1$. The tuple $e_{i,3}$ is inserted into the buffer; however, because ${}^iT$ is not updated, the releasing of tuples is not performed. When the tuple $e_{i,4}$ arrives, ${}^iT$ is updated to 5 and both the tuple $e_{i,3}$ and the tuple $e_{i,2}$ are released from the buffer. The arrival of the tuple $e_{i,5}$ and the arrival of the tuple $e_{i,6}$ trigger the release of the tuple $e_{i,4}$ and the release of the tuple $e_{i,5}$, respectively. The tuple $e_{i,7}$ is again an out-of-order tuple, and its delay is $delay(e_{i,7}) = 8 - 6 = 2$. Because $delay(e_{i,7})$ is larger than the buffer size K, which is 1 time unit, and the tuple whose timestamp is 7, i.e., the tuple $e_{i,5}$, has already been released, the tuple $e_{i,7}$ cannot be reordered correctly. As a result, $e_{i,7}$ is still an out-of-order tuple in the output stream of the K-slack buffer.

Algorithm 3.1 Buffer-based inter-stream disorder handling for m streams

```
1:  T^sync ← 0
2:  SyncBuf ← ∅
3:  for each tuple e from any stream S_i (i ∈ [1, m]) do
4:      if e.ts > T^sync then
5:          SyncBuf.insert(e)
6:          while SyncBuf has at least one tuple of each stream S_i (i ∈ [1, m]) do
7:              T^sync ← min{e'.ts | e' ∈ SyncBuf}
8:              Emit every e' that satisfies e'.ts = T^sync from SyncBuf
9:      else
10:         Emit e immediately
```

However, the delay of $e_{i,7}$ in the output stream of the buffer is reduced from 2 time units to 1 time unit.

Example 3.2.1 implies that to successfully reorder a tuple with a delay of k time units in a stream, a K-slack buffer of at least k time units is needed for the stream. Existing work that applies the K-slack algorithm to handle the intra-stream disorder either sets the buffer size K to a fixed value [Aba+03; Li+07], or increases K dynamically to be equal to the maximum delay among the so-far-observed out-of-order tuples [MP13a].

3.2.2 Handling Inter-Stream Disorder

The general idea of the typical buffer-based approach for handling the inter-stream disorder within m streams is to temporarily buffer tuples from each stream that is not the slowest one in terms of the timestamp progress (cf. Section 2.2.1), and release tuples from the buffer as the slowest stream progresses. Merging m timestamp-ordered streams into a single timestamp-ordered stream has been discussed in existing work such as [Gul+12]. However, as shown in Example 3.1.2, the inter-stream disorder could co-exist with the intra-stream disorder. Namely, each of the m input streams could also contain out-of-order tuples, if the intra-stream disorder within the stream is not handled completely.

In this dissertation, Algorithm 3.1 is used to handle the inter-stream disorder within multiple streams. This algorithm extends the existing buffer-based inter-stream disorder handling approach to consider the co-existence of the intra-stream disorder. Specifically, Algorithm 3.1 maintains a buffer to sort input tuples, and a variable T^{sync} to track the maximum timestamp among the tuples that have left the buffer. To be distinguished from a K-slack buffer, this buffer is referred to as a *synchronization buffer*. With Algorithm 3.1, an input tuple e can be processed in two different ways. If the tuple e satisfies $e.ts > T^{sync}$, then it is inserted into the synchronization buffer. Moreover, every tuple in the buffer that has the smallest timestamp is emitted from the buffer, as long as the buffer contains at least one tuple from each of the input streams (lines 4–8). If the tuple e does not satisfy $e.ts > T^{sync}$, it is then emitted immediately (lines 9–10). It can be observed that the size of the synchronization buffer is determined by the time skew (cf. Section 2.2.1) between

the fastest stream and the slowest stream—in terms of the timestamp progress—that are input to the buffer.

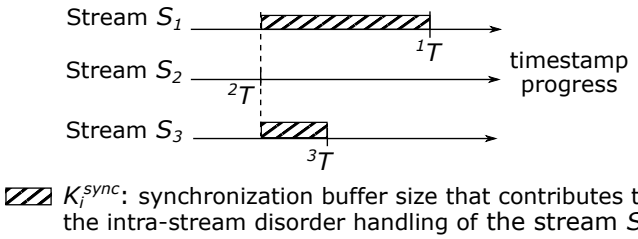


Figure 3.4: Intra-stream disorder handling performed implicitly by a synchronization buffer.

Note that a synchronization buffer can contribute to handling the intra-stream disorder within all input streams of the buffer but the slowest input stream. To illustrate this, let us consider the three streams in Figure 3.4. The stream S_1 is the fastest stream in terms of the timestamp progress; the stream S_3 is the second fastest; and the stream S_2 is the slowest. Using Algorithm 3.1 to handle the inter-stream disorder between these three streams, the value of the variable T^{sync} in the algorithm would be equal to the local current time ^2T of stream S_2. All so-far-arrived S_1 tuples whose timestamps are within the scope $(T^{sync}, {}^1T]$, and all so-far-arrived S_3 tuples whose timestamps are within the scope $(T^{sync}, {}^3T]$ are kept in and sorted by the synchronization buffer. Hence, the intra-stream disorder within the streams S_1 and S_3 is handled implicitly by a buffer of $^1T - T^{sync} = {}^1T - {}^2T$ time units and a buffer of $^3T - T^{sync} = {}^3T - {}^2T$ time units, respectively. Hereafter, this implicit buffer size within a synchronization buffer that contributes to handling the intra-stream disorder of a stream S_i is denoted by K_i^{sync}.

3.3 Buffer-Based Quality-Driven Disorder Handling (QDDH) Framework

Figure 3.5 describes the generic buffer-based QDDH framework proposed in this dissertation for processing continuous queries. Each query submitted to the DSPS is associated with a user-specified result-quality requirement, whose format depends on the type of the query. *K*-slack buffers, which are used for handling the intra-stream disorder, and synchronization buffers, which are used for handling the inter-stream disorder, are inserted as system operators (cf. Section 2.1.2) in the execution plan of a submitted query.

The framework adopts a *prior-to-operation* disorder handling strategy. Namely, for any query, the disorder within the input streams of the query is handled before the order-sensitive operators of the query are executed. For queries executed over a single input stream, only the intra-stream disorder handling is needed; whereas for queries executed over multiple input streams, both the intra-stream disorder handling and the inter-stream disorder handling are needed. A two-step disorder handling strategy is applied for queries with multiple input streams, and the intra-stream disorder is handled before the inter-stream disorder. When using query optimization techniques to optimize the execution of several concurrent queries, an

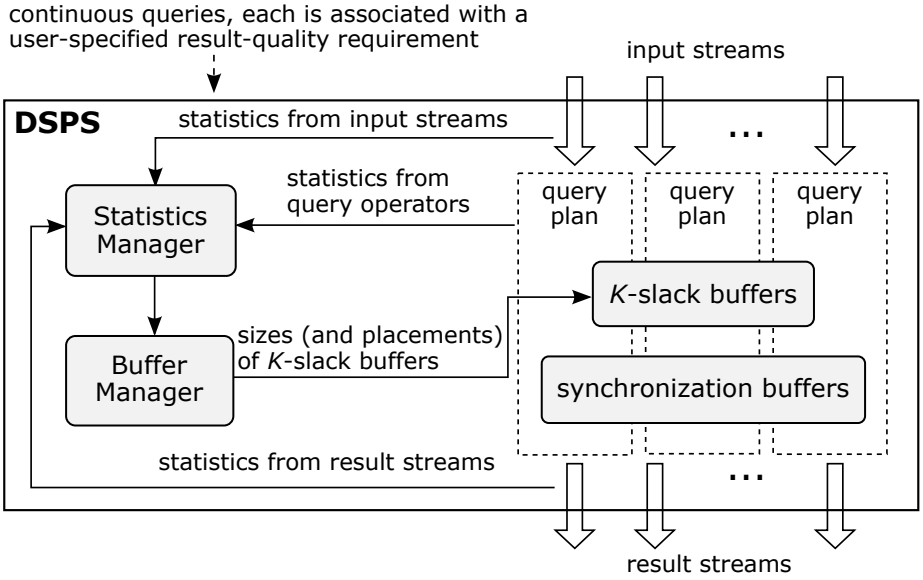


Figure 3.5: The generic buffer-based quality-driven disorder handling framework.

execution plan produced by the optimization process is indeed a *composite* plan that consists of multiple queries. The placement of *K*-slack buffers within such a composite query execution plan needs to be changed at the query runtime, if it is desired to perform QDDH at a minimum memory cost. Nevertheless, in the proposed QDDH framework, in any query execution plan, *K*-slack buffers are always applied earlier than synchronization buffers.

Note that when the disorder within an input stream S_i of a query is not handled completely, then the corresponding, derived stream of S_i that arrives at the order-sensitive operator of the query, denoted by S_i', would still contain out-of-order tuples. The QDDH framework applies the following policy to deal with these out-of-order tuples: if these tuples cannot contribute to computing any future results of the query, then they are discarded; otherwise, they are still forwarded to the order-sensitive query operator.

The concept of QDDH is implemented by two generic components in the framework: the *Buffer Manager* and the *Statistics Manager*. At the query runtime, the *Buffer Manager* determines dynamically the placement of *K*-slack buffers within the execution plan of query, as well as the size of each placed *K*-slack buffer. Its goal is to minimize the latency caused by disorder handling while honoring the user-specified result-quality requirement for the query. The new placements of buffers and the sizes of the placed *K*-slack buffers are determined based on statistics collected continuously by the *Statistics Manager*. Different statistics may be collected, including those collected from the input streams, those collected from the result streams, and those collected from the query operators.

Depending on the specific type of a query, or whether a query execution plan consists of multiple queries, the detailed behavior of the *Buffer Manager* and the statistics that need to be collected may be different. In this dissertation, three instantiations of the generic QDDH framework in Figure 3.5 will be discussed. They are

QDDH for individual SWA queries, QDDH for individual MSWJ queries, and QDDH for concurrent queries with shared operators. The first two instantiations will be presented in Chapter 4, and the third instantiation will be presented in Chapter 5.

3.4 Quality-Driven Buffer-Size Adaptation

A key task of the *Buffer Manager* in Figure 3.5 is to adapt the sizes of the *K*-slack buffers applied for a query at the query runtime in a quality-driven manner. This dissertation proposes an analytical-model-based buffer-size adaptation method to fulfill this task for queries with time-based windows. The core of this method is a modeling approach that directly captures the relation between the *K*-slack buffer sizes applied for a query and the consequent query-result quality. In the following, Section 3.4.1 introduces the theoretical foundation of this method. For the purpose of comparison, Section 3.4.2 presents the basic idea of a buffer-size adaptation method that is based on the usage of a proportional-derivative (PD) controller [Lev11]. Essentially, this control-based adaptation method treats the relation between the applied *K*-slack buffer sizes and the consequent query-result quality as a black box. Compared with such a control-based adaptation method, the analytical-model-based adaptation method proposed in this dissertation allows searching for the optimal *K*-slack buffer sizes to meet the result-quality requirement for a query in each iteration of the buffer-size adaptation.

3.4.1 Analytical-Model-Based Buffer-Size Adaptation

From Example 3.1.1 and Example 3.1.2, it can be observed that one key factor that determines the result quality of window-based operators is the *coverage*, i.e., the degree of completeness, of each instantaneous window over which the corresponding query results are produced. Formally, the coverage of an instantaneous window w at any moment in time is denoted by $Cvrg(w)$, and is defined as

$$Cvrg(w) = \frac{\text{\# tuples that are included in } w}{\text{\# tuples that would be included in } w \text{ if no stream disorder existed}} \tag{3.2}$$

Denoting the numerator and the denominator of Eq. (3.2) by $|w|$ and $|w|_{true}$, respectively, Eq. (3.2) can be shortened as

$$Cvrg(w) = \frac{|w|}{|w|_{true}} \tag{3.3}$$

Consider the sliding-window COUNT query in Example 3.1.1. If the stream disorder is not handled, then when the tuple $e_{i,8}$ ($e_{i,8}.ts = 10$) arrives, the coverage of the instantaneous window with scope $(0, 10]$ is 0.8, and the COUNT result for this instantaneous window produced at this moment is inaccurate. The two out-of-order tuples in this example, $e_{i,10}$ ($e_{i,10}.ts = 4$) and $e_{i,13}$ ($e_{i,13}.ts = 8$), fall into the scope of this instantaneous window as well. They are referred to as two *missing tuples* of this instantaneous window. Similarly, for the sliding-window join query in Example 3.1.2, at the arrival of the tuple c^3 from the stream S_2, the most recent instantaneous window over the stream S_1 contains the tuples A^1 and B^3. The coverage

of this instantaneous window is $\frac{2}{3} \approx 0.67$, because it has a missing tuple C^2. C^2 falls into the scope of this instantaneous window but has not arrived. As a result, the result tuple (C^2, c^3)[3] cannot be produced.

The definition of the coverage of an instantaneous window implies that a non-full coverage of an instantaneous window w (i.e., $Cvrg(w) < 1$) is caused by out-of-order tuples that have not arrived by the time the instantaneous window is constructed. If there is no stream disorder, then any instantaneous window constructed over any input stream should have a full coverage (i.e., $Cvrg(w) = 1$) at the moment it is constructed.

Handling the disorder within streams before forwarding them to a window-based query operator can increase the coverages of the instantaneous windows constructed over the streams. In general, the higher the coverages of the instantaneous windows, the higher the quality—in terms of, e.g., accuracy or completeness—of the query results produced over these instantaneous windows. For instance, when using a K-slack buffer of 5 time units to sort the stream in Example 3.1.1, the out-of-order tuple $e_{i,13}$ ($e_{i,13}.ts = 8$) will become an in-order tuple in the output stream of the buffer[3]. As a result, the coverage of the instantaneous window with scope $(0, 10]$ will increase from 0.8 to 0.9, and the produced COUNT result over this window is more accurate than the result that will be produced if the stream disorder is not handled. Moreover, when using a K-slack buffer of 7 time units, the disorder within the stream in Example 3.1.1 can be handled completely. The coverage of the instantaneous window with scope $(0, 10]$ will increase to 1 and the correct COUNT result for this window can be produced.

In general, it can be observed that, for a query with a window-based operator, (1) the coverages of the instantaneous windows constructed over each input stream of the query determines the result quality of the query, (2) the size of the K-slack buffer applied over an input streams directly influences the coverages of the instantaneous windows constructed over the stream, and (3) the buffer size K influences only the number of tuples that are actually included in an instantaneous window (i.e., $|w|$), but not the true number of tuples that belong to the window (i.e., $|w|_{true}$). Based on these observations, this dissertation proposes to analytically model the relation between the size of the K-slack buffer applied to an input stream and the number of tuples $|w|$ that would be included in an instantaneous window constructed over the stream, and then estimate the query-result quality based on $|w|$. In this way, the relation between the sizes of the K-slack buffers applied for a query and the consequent query-result quality can be captured by an analytical model. Based on this analytical model, one can directly determine the optimal K-slack buffer sizes needed to meet the user-specified result-quality requirement for the query.

The remainder of this subsection describes the method proposed in this dissertation for modeling the relation between the size of the K-slack buffer applied to an input stream and the number of tuples that would be included in an instantaneous window constructed over the stream. Descriptions on how to further model the relation between the applied K-slack buffer size and the consequent query-result quality is deferred to Chapter 4, in the respective sections that describe the instantiations of the generic QDDH framework for specific query types.

[3]Recall from Section 3.2 that a K-slack buffer of k time units can successfully handle out-of-order tuples whose delays are not larger than k time units.

For an input stream S_i, let D_i denote a discrete random variable that represents the coarse-grained delay of a tuple e_i in S_i. Specifically, let D_i take the value 0 if $delay(e_i) = 0$, take the value 1 if $delay(e_i) \in (0, g]$, take the value 2 if $delay(e_i) \in (g, 2g]$, and so forth; g is a configurable parameter in the proposed QDDH framework and is referred to as the *K-search granularity*. Furthermore, let f_{D_i} denote the probability density function (PDF) of D_i, i.e., $f_{D_i}(d) = Pr[D_i = d]$, where $d \in \{0, 1, 2, \dots\}$. Based on the assumption that the near future resembles the recent past, for each input stream S_i, the *Statistics Manager* in Figure 3.5 monitors the delays of the input tuples that are within a window R_i^{stat} over the stream S_i's recent history, and approximates f_{D_i} using a histogram $\mathcal{H}_i$, which maintains the statistics of the monitored tuple-delays in S_i. Without a priori knowledge of the disorder pattern within the stream, it is difficult to find a fixed size for the window R_i^{stat}. Moreover, such a fix-sized R_i^{stat} is sensitive to varying disorder patterns. Hence, in this dissertation, the approach proposed in [BG07] is used to dynamically adapt the size of the window R_i^{stat}, based on the rate of changes detected from the delays of tuples in the window R_i^{stat} itself. The sizes of the R^{stat} windows for different input streams are adapted separately.

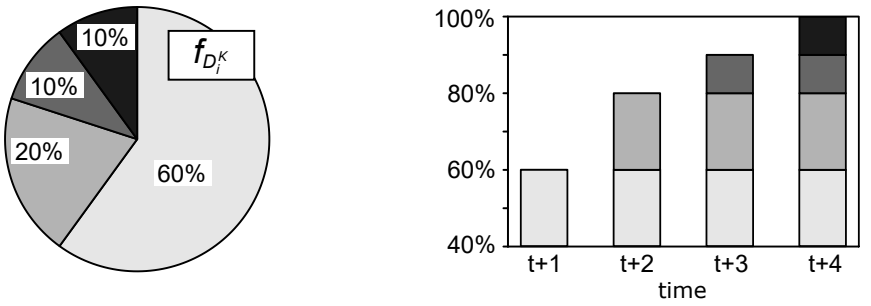
Note that f_{D_i} captures the tuple-delay characteristics within an input stream S_i. After the intra-stream disorder handling by a K-slack buffer, and potentially the inter-stream disorder handling by a synchronization buffer, the tuple-delay characteristics in the corresponding, derived stream S_i' that is received by the window-based query operator is often different from f_{D_i}. Let D_i^K denote a discrete random variable representing the coarse-grained delay of a tuple in S_i' under a certain value of the buffer size K. Let $f_{D_i^K}$ represent the PDF of D_i^K. The change from f_{D_i} to $f_{D_i^K}$ can be captured based on the following observation: For any tuple e_i in an input stream S_i, the delay of the tuple e_i within the corresponding, derived stream received by the window-based operator changes from $delay(e_i)$ to $delay^K(e_i)$, where $delay^K(e_i) = max\{0, delay(e_i) - K - K_i^{sync}\}$, and K_i^{sync} is the implicit buffer size within a synchronization buffer that contributes to the intra-stream disorder handling of S_i (cf. Section 3.2). Hence, $f_{D_i^K}$ can be derived from f_{D_i} using Eq. (3.4).

$$f_{D_i^K}(d) = \begin{cases} \sum_{d'=0}^{(K+K_i^{sync})/g} f_{D_i}(d'), & d = 0 \\ f_{D_i}(d + \frac{K+K_i^{sync}}{g}), & d \in \{1, 2, 3, \dots\} \end{cases} \tag{3.4}$$

Again, based on the assumption that the near future resembles the recent past, in the proposed QDDH framework, K_i^{sync} is estimated as $\overline{K_i^{sync}} - min\{\overline{K_j^{sync}} | j \in [1, m]\}$, where $\overline{K_i^{sync}}$ represents the average of all measurements of K_i^{sync} that are collected by the *Statistics Manager* within the window R_i^{stat}[4]. With the estimated K_i^{sync}, the histogram $\mathcal{H}_i$ that is used to approximate f_{D_i} can then be used to approximate $f_{D_i^K}$ as well.

Now the relation between the K-slack buffer size K_i and the number of tuples that would be included in an instantaneous window constructed over the corresponding, disorder-handled, derived stream S_i', i.e., $|w_i|$ in Eq. (3.3), can be modeled by estimating $|w_i|$ based on $f_{D_i^K}$. This estimation exploits the following observation: If an

[4]According to Algorithm 3.1, K_i^{sync} is determined by the local current times of the output streams of the K-slack buffers. However, it will be shown in Section 4.2.3 that under the two-step disorder handling strategy described in Section 3.3, one can determine K_i^{sync} based on the local current times of the original input streams.



(a) Monitored $f_{D_i^K}$ at ${}^iT' = t$

(b) Expected percentage of received tuples whose timestamps fall into (t, t+1]

Figure 3.6: The expected percentage of so-far-received tuples belonging to a certain time unit in a stream among all tuples belonging to the time unit in the stream under a certain distribution of tuple delays in the stream.

instantaneous window is divided into small segments, then a recent segment of the window (i.e., a segment closer to the upper endpoint of the instantaneous window) often has more missing tuples than an old segment of the window. The reason is that, by the time the instantaneous window is constructed, out-of-order tuples whose timestamps fall into the scope of an old segment of the window might have arrived already, and have been inserted into the window; whereas out-of-order tuples whose timestamps fall into the scope of a recent segment of the window can be observed only at later points in time.

The above observation can be illustrated by Figure 3.6. Let ${}^iT'$ denote the local current time of the derived stream S'_i, i.e., the maximum timestamp among the so-far-observed tuples in the stream S'_i. Assume that at the moment of ${}^iT' = t$ ($t \in \mathbb{T}$), the monitored $f_{D_i^K}$ of S'_i is as shown in Figure 3.6a. If the value of the K-slack buffer size K is not adjusted during the following time unit $(t, t+1]$, then based on the current $f_{D_i^K}$, one can estimate that, among tuples whose timestamps fall into the time unit $(t, t+1]$, 60% of them will arrive in order, 20% of them will be delayed by 1 time unit, 10% of them will be delayed by 2 time units, and another 10% of them will be delayed by 3 time units. From another perspective, this means that one can expect to receive only 60% of the tuples whose timestamps are within the time unit $(t, t+1]$ by the time of $t+1$ (cf. Figure 3.6b). By the time of $t+2$, one can expect to receive 80% of the tuples belonging to the time unit $(t, t+1]$; because by that time, the 20% of the tuples that are delayed by 1 time unit should have arrived. Finally, by the time of $t+4$, one can expect to receive all tuples belonging to the time unit $(t, t+1]$. Figure 3.6b shows that, for a specific time unit, the older it becomes, the more likely that all out-of-order tuples falling into the scope of this time unit have been observed by the current time. This also means that at any specific point in time, an old time unit is more likely to have observed all out-of-order tuples falling into its scope than a recent time unit.

To capture the difference in the coverages of different segments of an instantaneous window w_i constructed over a stream S_i, this dissertation adopts the notion of *basic window*, which was introduced by Gedik et al [Ged+07]. Specifically, an instantaneous window is divided into basic windows, each of which has a size of b time

units. The instantaneous window consists of $n_i = \lceil W_i/b \rceil$ basic windows, where W_i is the size of the sliding window over the stream S_i (cf. Section 2.1.2). Let w_i^l denote the l-th, $l \in [1, n_i]$, basic window of w_i; w_i^1 represents the most recent basic window of w_i.

Let $w_{i,\top}$ denote the most recent instantaneous window constructed over the stream S_i'. The upper endpoint of $w_{i,\top}$ is ${}^iT'$. The scope of each basic window $w_{i,\top}^l$ of $w_{i,\top}$ can be determined as $({}^iT' - l \cdot b, {}^iT' - (l-1) \cdot b]$ for $l \in [1, n_i - 1]$, and $({}^iT' - W_i, {}^iT' - (n_i - 1) \cdot b]$ for $l = n_i$. For the basic window $w_{i,\top}^1$, among all S_i' tuples e_i whose timestamps fall into the scope of $w_{i,\top}^1$, tuples that have no delays, i.e., $delay^K(e_i) = 0$, should have arrived. Hence, the expected number of tuples that are included in $w_{i,\top}^1$, denoted by $|w_{i,\top}^1|$, can be estimated as $|w_{i,\top}^1| = r_i \cdot b \cdot f_{D_i^K}(0)$. Recall from Section 2.1.1 that r_i represents the tuple arrival rate of the stream S_i. For the basic window $w_{i,\top}^2$, all tuples whose delays satisfy $delay^K(e_i) \in [0, \frac{b}{g}]$ should have arrived, and $|w_{i,\top}^2|$ can be estimated as $r_i \cdot b \cdot \sum_{d=0}^{\frac{b}{g}} f_{D_i^K}(d)$. In general, for any basic window $w_{i,\top}^l$, $l \in [1, n_i]$, $|w_{i,\top}^l|$ can be estimated using Eq. (3.5).

$$|w_{i,\top}^l| = \begin{cases} r_i \cdot b \cdot \sum_{d=0}^{\frac{(l-1)b}{g}} f_{D_i^K}(d), & l \in [1, n_i - 1] \\ r_i \cdot (W_i - (n_i - 1) \cdot b) \cdot \sum_{d=0}^{\frac{(n_i-1)b}{g}} f_{D_i^K}(d), & l = n_i \end{cases} \tag{3.5}$$

Finally, the number of tuples included in the entire instantaneous window, $|w_{i,\top}|$, can be estimated as

$$\begin{aligned} |w_{i,\top}| &= \sum_{l=1}^{n_i} |w_{i,\top}^l| \\ &= r_i \cdot \left(b \cdot \sum_{l=1}^{n_i - 1} \sum_{d=0}^{\frac{(l-1)b}{g}} f_{D_i^K}(d) + (W_i - (n_i - 1) \cdot b) \cdot \sum_{d=0}^{\frac{(n_i-1)b}{g}} f_{D_i^K}(d) \right) \end{aligned} \tag{3.6}$$

Note that a bigger basic-window size b implies a more conservative estimation of $|w_{i,\top}|$ than a smaller b. When the value of b is chosen such that $n_i = 1$ for all $i \in [1, m]$, it means that the estimation of $|w_{i,\top}|$ considers only in-order tuples.

3.4.2 Control-based Buffer-Size Adaptation

Other than the analytical method described in Section 3.4.1, the relation between the K-slack buffer sizes applied for a query and the consequent query-result quality can also be treated as a black box. The input to this black box is the K-slack buffer sizes and the output is the query-result quality. The problem of adjusting the K-slack buffer sizes to meet the user-specified result-quality requirement can be well mapped to a *control* problem. The key idea is using a feedback loop to control the behavior of a system or a process by comparing the output of the system or process, which is called the *process variable* (PV), to a desired value, which is called the *setpoint* (SP), and applying the difference between the measured process variable and the setpoint as an *error* signal to dynamically change the system or process to make the measured process variable get closer to the setpoint [Lev11].

As a comparison to the proposed analytical-model-based buffer-size adaptation method, this dissertation also studies how to perform buffer-size adaptation using a *proportional-derivative (PD) controller*—a variant of the well-known *proportional-integral-derivative (PID) controller*. This subsection describes the general idea of this control-based buffer-size adaptation method. Instantiations of this method for specific query types, especially the choices of the setpoint, will be discussed in the respective sections in Chapter 4.

In general, for a chosen setpoint, the control action of a PID controller is generated as a weighted sum of three terms: a proportional term (P) that accounts for the present control error, an integral term (I) that accounts for a summary of past errors, and a derivative term (D) that accounts for a prediction of future errors. The weights of the three terms are denoted by U_p, U_i, and U_d, respectively. A PD controller does not have the integral term. The PD controller is chosen in this dissertation because the integral term can cause the so-called *integral windup*, if the measured process variable keeps above the setpoint for an extended period. In the context of quality-driven buffer-size adaptation, the problem of integral windup can occur easily during periods when the input streams have no, or little, disorder. During these periods, the produced query-result quality may stay well above the user-specified result-quality requirement, even though the K-slack buffer sizes are reduced to zero.

Formally, let $u(j)$ denote the control action of the j-th iteration of the buffer-size adaptation for a query, and let $err(j)$ denote the present control error. The PD controller determines $u(j)$ as

$$u(j) = U_p \cdot err(j) + U_d \cdot \frac{\mathrm{d}err(j)}{\mathrm{d}j}. \tag{3.7}$$

Recall from Section 3.2.1 that to remove the intra-stream disorder within a stream S_i completely, the required K-slack buffer size is determined by the maximum tuple delay within the stream. Let $MaxD_i$ denote this maximum tuple delay. Namely, the buffer size that can be applied over the stream S_i is within the range $[0, MaxD_i]$[5]. Due to the unbounded and dynamic nature of data streams, there is often no a priori knowledge of the true value of $MaxD_i$. Hence, in practice, $MaxD_i$ is estimated by the maximum delay among *so-far-arrived* tuples in S_i. As a result, the range $[0, MaxD_i]$ may extend over time, as out-of-order tuples with larger delays are observed.

To bound the buffer size produced in each control-based adaptation iteration within the range $[0, MaxD_i]$, and to account for the potential extension of this range at the query runtime, this dissertation proposes to apply control actions produced by Eq. (3.7) not directly on the parameter K_i. Instead, a parameter α ($\alpha \in [0, 1]$) is introduced and the buffer size K_i is rewritten as $\alpha \cdot MaxD_i$. At the query runtime, $MaxD_i$ can be updated dynamically, and the control actions are applied to the parameter α.

Figure 3.7 shows the generic PD-controller-based buffer-size adaptation method applied in this dissertation. The output of the controller is the adjustment, i.e., the increase or the decrease, of the parameter α, denoted by $\Delta\alpha$; The final value of α produced in the j-th adaptation iteration is $\alpha(j) = max\{0, min\{\alpha(j-1) + \Delta\alpha(j), 1\}\}$.

[5] A buffer size that is larger than $MaxD_i$ is possible but is unnecessary.

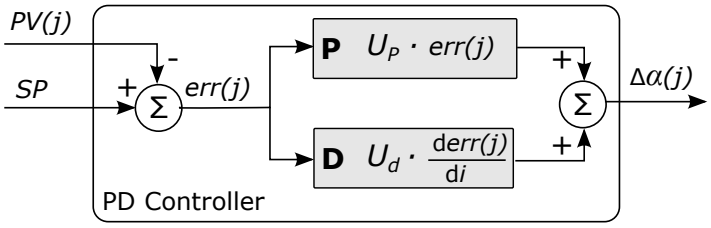


Figure 3.7: Control-based *K*-slack buffer-size adaptation using a PD controller.

3.5 Related Work

This section discusses in detail the four categories of disorder handling approaches mentioned in Section 2.2.2, as well as the related work on load shedding, which shares certain similarity with disorder handling.

3.5.1 Disorder Handling Approaches

Buffer-based Disorder handling

The basic mechanism of buffer-based disorder handling has been described in Section 3.2. For handling the intra-stream disorder, Aurora [Aba+03], Cayuga [Dem+07], and the work of [Li+07] fixed the *K*-slack buffer size applied for a query during the query runtime. In contrast, Babu et al. [BSW04] and Mutschler et al. [MP13a] proposed methods to automatically adapt the applied buffer sizes at runtime to react to the changing disorder characteristics in the streams, so that the intra-stream disorder can be handled as completely as possible.

For handling the inter-stream disorder, the approach of using buffers to synchronize input streams before executing an order-sensitive operator has been applied in many existing systems such as StreamCloud [Gul+12] and BiStream [Lin+15]. Focusing on time-based sliding-window join queries, the work of [HAE05] compared the approach that enforces an ordered processing of input tuples with the approach that allows out-of-order processing of input tuples but enforces an ordered release of result tuples, and studied the memory consumption and the response time of the two approaches for 2-way joins.

Wu et al. [WTZ07] considered the presence of both the intra-stream disorder and the inter-stream disorder for join queries without windows. They followed the two-step disorder handling strategy as this dissertation does (cf. Section 3.3).

In contrast to this dissertation, none of the existing buffer-based disorder handling approaches have considered minimizing the latency caused by buffering in a quality-driven manner.

Punctuation-based Disorder Handling

Punctuation-based disorder handling approaches [Kri+10; Li+05b; Li+08; Liu+09; SW04a; Tuc+03; Aki+13] rely on special tuples within data streams, called *punctuations*, to indicate that no future tuples with timestamps smaller than the timestamp of a punctuation are expected. When a punctuation is received, a window operator can determine the instantaneous windows for which no future out-of-order tuples

are expected, and produce query results for those instantaneous windows. *Heartbeats* used in [SW04a; KS09] and *partial order guarantees* used in [Liu+09] are special types of punctuations.

Punctuations explicitly inform query operators when to return results for instantaneous windows; as a result, the query operators can process out-of-order input streams directly. However, the quality of the produced query results is fundamentally limited by the quality of the punctuations [Kri+10]. Most existing work in this area focused more on the usage, rather than the generation, of punctuations. It is assumed that punctuations are either provided by external data sources, or can be generated easily by a DSPS based on a priori knowledge about the application semantics or the disorder characteristics within the input streams. However, this assumption does not hold in many real-world scenarios. In those scenarios, the method proposed in [SW04a] for generating heartbeats can be used to generate punctuations. This method is based on runtime-estimated parameters that capture the skews between streams, the disorder within streams, and the latency in streams reaching a DSPS.

An out-of-order tuple that is missing from an instantaneous window can be detected only after the window has been constructed; otherwise, the tuple is not a missing tuple of the instantaneous window (cf. Section 3.4.1). This fact implies that the punctuation indicating the receipt of all out-of-order tuples falling into the scope of an instantaneous window cannot be generated before all these tuples have been observed. Therefore, punctuation-based disorder handling approaches share the same latency issue as buffer-based disorder handling approaches; and the tradeoff between the end-to-end latency and the quality of query results still exists. The QDDH concept proposed in this dissertation can be applied to do quality-driven punctuation generation. Specifically, instead of producing a punctuation for an instantaneous window until all out-of-order tuples belonging to the window have been received, the punctuation can be produced earlier, as soon as the quality of the produced query results can meet the user-specified requirements.

Speculation-based Disorder Handling

Buffer-based and punctuation-based disorder handling approaches can be further categorized as *conservative* disorder handling approaches, because they both, to a certain extent, wait for out-of-order tuples to avoid degrading the quality of produced query results. In contrast, speculation-based disorder handling approaches [Bar+07; Bri+08; Liu+09; Ryv+06; MC08] are *aggressive* disorder handling approaches. They assume an in-order arrival of stream tuples, and produce the result of an instantaneous window immediately when the window is constructed. To deal with the result-quality issue caused by out-of-order tuples, they use the technique of *revisions (a.k.a., retractions)*. Specifically, when an out-of-order tuple e is detected, previously-produced query results that are affected by the tuple e are invalidated, and new revisions of those results are produced by taking e into account of the re-computation.

Speculative disorder handling does not delay the result computation of instantaneous windows. However, computing result revisions requires maintaining a certain history of the input streams or the result stream. Moreover, retracting and re-computing query results complicate the logic of query operators, and often lead to expensive operator implementations regarding the CPU consumption. For highly out-of-ordered data streams, one query result may be revised multiple times before the final

exact revision is produced. This may exhaust the CPU and may cause a high latency as well. In addition, speculative computation of query-results also requires the application that consumes the query results to be able to interpret and deal with result revisions.

The QDDH concept proposed in this dissertation can be applied in combination with speculation as well. Specifically, the QDDH concept can help reducing the overhead of producing result revisions by stopping producing more revisions of a result when the quality of the latest revision already meets the user-specified requirement.

Hybrid Disorder Handling

Krishnamurthy et al. [Kri+10] applied speculation in combination with punctuations, to deal with query-result quality issues caused by inaccurate punctuations. Specifically, they proposed to treat out-of-order tuples that appear after the corresponding punctuations as separate data partitions, and process those partitions independently. Partial results of those partitions can be merged with previously produced, inaccurate or incomplete query results on demand, so as to preserve the integrity of stream histories. This approach assumes that the entire history of the input streams and the query results are persisted in, for example, a database.

Mutschler et al. [MP13b] proposed to combine speculation and buffering for processing pattern-detection queries. To reduce the latency of the pattern detection, input tuples are speculatively released from the K-slack buffer (cf. Section 3.2.1) applied for the input stream of a query. The speculation degree is controlled by a parameter α, which is adjusted at the query runtime based on the CPU load on the system. Note that the control-based buffer-size adaptation method introduced in Section 3.4.2 also uses a parameter α to control the premature release of tuples from a K-slack buffer; however, different from the work of Mutschler et al., the parameter α in this dissertation is adjusted based on the quality of produced query results.

3.5.2 Load Shedding

Load shedding in the context of data stream processing describes the technique of processing a selected portion of input tuples when the available system resources (e.g., CPU and memory) cannot match the resource demand imposed by the query. Load shedding has been studied extensively in the literature [Tat+03; BDM04; Ged+07; LZ08; MZ10]. Three key questions that need to be answered are when to shed load, where to shed load, and how much load to shed, so that a certain QoS target (e.g., end-to-end latency, utility[Tat+03], result accuracy [BDM04; LZ08], or output rate [Ged+07]) can be met.

Handling disorder within data streams shares similarity with load shedding, because the effect of a tuple dropped during load shedding on the query-result quality is the same as the effect of an unsuccessfully-handled out-of-order tuple during disorder handling on the query-result quality. However, in load shedding, a DSPS can control the amount of tuples to be excluded from instantaneous windows directly by, for instance, defining a certain sampling rate. In contrast, in disorder handling, the amount of missing tuples in an instantaneous window is not directly controllable, but is determined by both the applied K-slack buffer size and the disorder situation within the window.

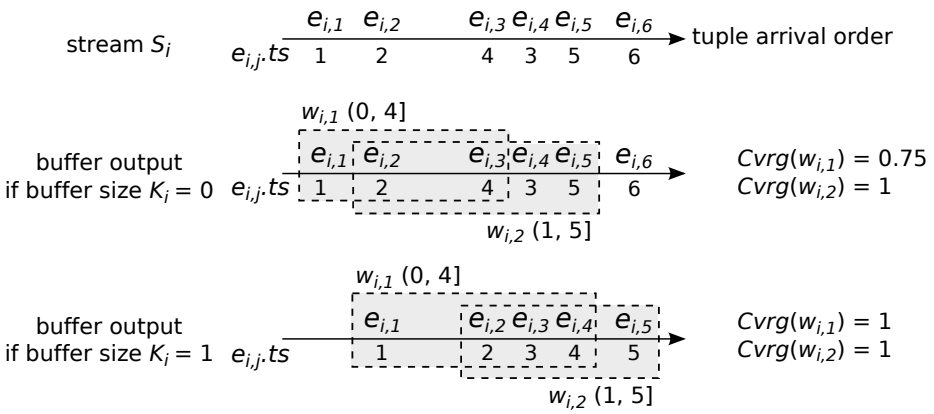


Figure 3.8: Influence of the K-slack buffer size and the disorder characteristics on the coverages of instantaneous windows.

Example 3.5.1. Consider the example in Figure 3.8. Assume that a sliding window of 4 time units is applied over the input stream S_i. For the first instantaneous window $w_{i,1}$, whose scope is $(0, 4]$, the coverage is $Cvrg(w_{i,1}) = 0.75$ if the size of the K-slack buffer used to sort S_i is $K_i = 0$. If the buffer size is $K_i = 1$, then the coverage of $w_{i,1}$ is $Cvrg(w_{i,1}) = 1$. In contrast, for the instantaneous window $w_{i,2}$, whose scope is $(1, 5]$, the coverage is $Cvrg(w_{i,2}) = 1$ for both $K_i = 0$ and $K_i = 1$.

Moreover, in load shedding, to determine which tuples to drop, a DSPS needs to consider only the contribution of a tuple to the query-result quality based on the value of the tuple. In contrast, in QDDH, to determine which out-of-order tuple can be handled unsuccessfully to trade high query-result quality for low latency, a DSPS needs to consider not only the value but also the delay of a tuple. Namely, one more dimension needs to be considered in QDDH than in load shedding.

3.6 Summary

This chapter started to address the research question of how to provide a user-configurable tradeoff between the performance and the query-result quality when facing data imperfection within data streams, particularly, the stream disorder problem. Disorder handling involves an inevitable tradeoff between the end-to-end latency and the quality of produced query results. This chapter gave an overview of the buffer-based, quality-driven disorder handling framework proposed in this dissertation. The framework aims to minimize the latency incurred by disorder handling while honoring user-specified result-quality requirements.

Focusing on aggregate queries and join queries with time-based sliding windows, this chapter also described the essence of an analytical-model-based method, and a control-based method, which are used to perform quality-driven buffer-size adaptation in the proposed QDDH framework. Compared with the control-based adaptation method, the analytical-model-based method can capture the relation between the buffer sizes applied for disorder handling and the consequent query-result quality directly, thereby allowing the method to search for the optimal buffer sizes to

meet the user-specified result-quality requirements at each iteration of the buffer-size adaptation.

The generic QDDH framework as well as the two buffer-size adaptation methods described in this chapter will be instantiated for specific query types in Chapter 4 and Chapter 5.

4

Quality-Driven Disorder Handling for Individual Queries

This chapter describes two instantiations of the generic buffer-based QDDH framework introduced in Chapter 3. One instantiation is for sliding-window aggregate (SWA) queries (Section 4.1), and the other one is for m-way sliding-window join (MSWJ) queries (Section 4.2). Both instantiations focus on time-based sliding windows. For each instantiation, the respective section describes the metric applied for measuring the quality of produced query results, as well as the instantiations of the components in the generic QDDH framework (cf. Figure 3.5). The experimental evaluation of the two instantiations is presented in Section 4.3.

4.1 QDDH for Sliding-Window Aggregate Queries

4.1.1 Result-Quality Metric

Accuracy in terms of the *relative error* is a widely-adopted result-quality metric for aggregate queries [BDM04; TXB06; CKT08; MZ10; Aga+13]. Specifically, let $\hat{A}$ denote a produced aggregate result, and A the corresponding exact aggregate result that would be produced if the input streams did not contain disorder, the relative error ϵ of the produced aggregate result is defined as $\epsilon = \frac{|A-\hat{A}|}{|A|}$. A user specifies the result-quality requirement for a SWA query by specifying a threshold for the relative errors in produced aggregate results. The error threshold is in the form of (ϵ_{thr}, δ), $\delta \in (0,1)$, which states that the relative error in a query result exceeds ϵ_{thr} with probability at most δ. The parameter δ is referred to as the *confidence level*.

4.1.2 QDDH-Framework Instantiation Overview

Figure 4.1 gives an overview of the proposed instantiation of the buffer-based QDDH framework for individual SWA queries. A SWA query has only one input stream, and hence only the intra-stream disorder handling, using a K-slack buffer, is needed. Without loss of generality, let us consider a DSPS implementation that maps the logical window operator and the aggregate operator to one physical SWA operator (cf. Section 2.1.2).

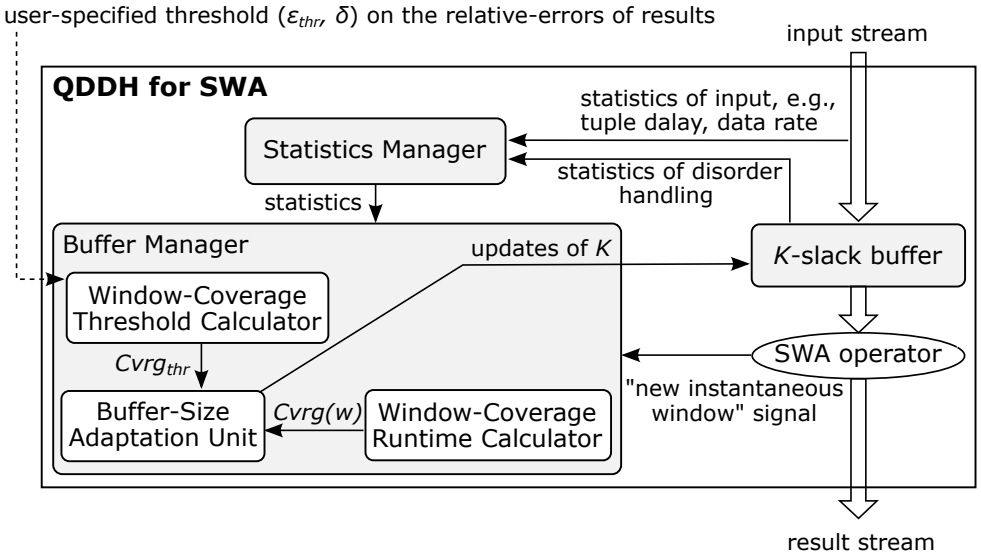


Figure 4.1: Instantiation of the buffer-based QDDH framework for individual SWA queries.

The quality-driven buffer-size adaptation is performed by the *Buffer Manager*, which consists of three sub-components: a *Window-Coverage Threshold Calculator*, a *Window-Coverage Runtime Calculator*, and a *Buffer-Size Adaptation Unit*. The *Statistics Manager* collects and maintains statistics that are needed by all three sub-components of the *Buffer Manager*.

The runtime behavior of the *Buffer Manager* is sketched in Algorithm 4.1. Each time the *Buffer Manager* receives a signal indicating that a new instantaneous window has been constructed, it first checks whether a predefined warm-up period L_{warmup} has passed. The warm-up period is used to allow the *Statistics Manager* to collect enough statistics for making buffer-size adaptation decisions. If the warm-up period has passed, the *Buffer Manager* then invokes the *Window-Coverage Threshold Calculator* to translate the user-specified result relative-error threshold (ϵ_{thr}, δ) to a *window-coverage threshold* $Cvrg_{thr}$ (cf. Section 4.1.3), and invokes the *Window-Coverage Runtime Calculator* to measure the actual coverages of previously-constructed instantaneous windows (cf. Section 4.1.4). The obtained window-coverage threshold $Cvrg_{thr}$ and window-coverage measurements *Cvrg* are fed into the *Buffer-Size Adaptation Unit*, which determines a new value of the buffer size *K* using a certain buffer-size adaptation method (cf. Section 4.1.5 and Section 4.1.6).

4.1.3 Calculating Window-Coverage Threshold

Example 3.1.1 and the discussions at the beginning of Section 3.4.1 imply that the coverages of instantaneous windows can be used as an intermediate metric for measuring the quality of the results of aggregate queries. Computing aggregates over incomplete instantaneous windows constructed under incomplete intra-stream disorder handling is essentially approximate query processing (AQP) over a sample of the original input tuples. Existing work in the area of sampling-based AQP [Aga+13;

Algorithm 4.1 The behavior of the *Buffer Manager* in the QDDH-framework instantiation for individual SWA queries (cf. Figure 4.1)

Input: specification of a SWA query
(ϵ_{thr}, δ) - user-specified result relative-error threshold
L_{warmup} - length of the warm-up period
"new instantaneous window" signals
statistics from the *Statistics Manager*
Output: new settings of the K-slack buffer size
1: **for each** "new instantaneous window" signal **do**
2: **if** $warmupPeriodHasPassed(L_{warmup})$ **then**
3: Invoke the *Window-Coverage Threshold Calculator* to calculate a window-coverage threshold $Cvrg_{thr}$.
4: Invoke the *Window-Coverage Runtime Calculator* to measure the coverages $Cvrg$ of previously-constructed instantaneous windows.
5: Taking $Cvrg_{thr}$ and $Cvrg$ obtained above as input, invoke the *Buffer-Size Adaptation Unit* to determine the new buffer size K to be applied before the next adaptation iteration.

LZ08; MZ10] has shown that one can use statistical inequalities and the *central limit theorem* to build an error model for the results of an aggregate query produced under sampling. Such an error model relates the sampling rate p to the relative error ϵ in a produced aggregate result. Hence, given a relative-error threshold, one can derive the minimum sampling rate required to meet the threshold. For each instantaneous window constructed over the input stream of a SWA query, the sampling rate p determines the proportion of tuples within the window to be retained for processing; thus, there is a direct semantic mapping between the sampling rate and the window coverage. This observation inspires us to build the error model for the aggregate results produced under disorder handling in a similar way as for the aggregate results produced under sampling, and then to derive a window-coverage threshold, i.e., the minimum coverage that an instantaneous window must reach to not violate the user-specified result relative-error threshold.

The *Window-Coverage Threshold Calculator* in Figure 4.1 is responsible for calculating the window-coverage threshold for a given user-specified error threshold for a SWA query. The specific calculation depends on the aggregate function. Let us take the SUM function as an example. Denote the set of tuples belonging to an instantaneous window in the absence of stream disorder as Z, the number of tuples in Z (i.e., $|Z|$) as N, the values of the tuples to be summed up as $z_1, z_2, \ldots, z_N$, and the exact SUM result over Z as $A = \sum_{i=1}^{N} z_i$. Now assume that the stream disorder is present. Let us denote the set of tuples that are actually included in the instantaneous window after disorder handling by the time the window is constructed as Z'. One can observe that Z' is a subset of Z. Without knowing which tuples in Z are absent in Z', let us assume that the probability that a tuple in Z gets included in Z' is P. Based on this assumption, one can estimate that the expected number of tuples in Z' is $|Z'| = N \cdot P$. Hence, the coverage of the instantaneous window is $Cvrg = \frac{|Z'|}{|Z|} = \frac{N \cdot P}{N} = P$. Namely, the value of P equals the value of $Cvrg$.

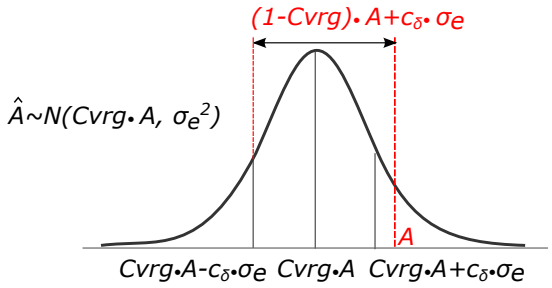


Figure 4.2: Schematic illustration of the distribution of a produced SUM result $\hat{A}$ by the central limit theorem. (Assume that the corresponding exact result $A > 0$.)

Based on the above value-equivalence between P and $Cvrg$, N random variables $X_1, X_2, \ldots, X_N$ are defined, such that $X_i = z_i$ $(i \in [1, N])$ with probability $Cvrg$ and $X_i = 0$ with probability $1 - Cvrg$. The sum of these random variables is $\hat{A} = \sum_{i=1}^{N} X_i$, which is indeed the SUM result over Z'. Assume that the mean and the variance of the tuples in Z are $\mu = \sum_{i=1}^{N} z_i/N$ and $\sigma^2 = \sum_{i=1}^{N}(z_i - \mu)^2/N$, respectively. The mean and the variance of the random variable $\hat{A}$ can be derived as follows:

$$E[\hat{A}] = \sum_{i=1}^{N} E[X_i] = \sum_{i=1}^{N} (z_i \cdot Cvrg + 0 \cdot (1 - Cvrg)) = Cvrg \cdot A,$$

$$\begin{aligned} Var[\hat{A}] &= E[\hat{A}^2] - E[\hat{A}]^2 = E[\hat{A}^2] - (Cvrg \cdot A)^2 \\ &= \sum_{i=1}^{N} E[X_i^2] + \sum_{1 \le i \ne j \le N} E[X_i]E[X_j] - (Cvrg \cdot A)^2 \\ &= \sum_{i=1}^{N} (Cvrg \cdot z_i^2) + \sum_{1 \le i \ne j \le N} (Cvrg^2 \cdot z_i \cdot z_j) - (Cvrg \cdot \sum_{i=1}^{N} z_i)^2 \\ &= (Cvrg - Cvrg^2) \sum_{i=1}^{N} z_i^2 \\ &= (Cvrg - Cvrg^2) \frac{\sigma^2 + \mu^2}{N \cdot \mu^2} A^2. \end{aligned}$$

By the central limit theorem, one can assume that $\hat{A}$ is normally distributed with mean $Cvrg \cdot A$ and standard deviation $\sigma_e = \sqrt{(Cvrg - Cvrg^2)\frac{\sigma^2+\mu^2}{N \cdot \mu^2}} \cdot |A|$, where $|A|$ represents the absolute value of A. Based on the statistical properties of the normal distribution, one can find the associated *critical value* c_δ for a given confidence level δ $(0 < \delta < 1)$. The critical value c_δ for a given δ means that the probability that $\hat{A}$ lies outside the range $[Cvrg \cdot A - c_\delta \cdot \sigma_e, Cvrg \cdot A + c_\delta \cdot \sigma_e]$ is at most δ (cf. Figure 4.2). Therefore, to guarantee that the relative error of $\hat{A}$ exceeds ϵ_{thr} with probability at most δ, the value of $Cvrg$ must satisfy the condition in Eq. (4.1).

$$\begin{aligned} &\frac{(1 - Cvrg) \cdot |A| + c_\delta \cdot \sigma_e}{|A|} \\ =&(1 - Cvrg) + c_\delta \cdot \sqrt{(Cvrg - Cvrg^2) \cdot \frac{\sigma^2 + \mu^2}{N \cdot \mu^2}} \le \epsilon_{thr}, \end{aligned} \tag{4.1}$$

Algorithm 4.2 Calculate the window-coverage threshold for a sliding-window SUM query

Input: W - window size
(ϵ_{thr}, δ) - user-specified result relative-error threshold
I_{recalc} - interval for re-calculating $Cvrg_{thr}$
Output: $Cvrg_{thr}$ - window-coverage threshold

1: **if** $isFirstCalculation() \vee needRecalculation(I_{recalc})$ **then**
2: Calculate N in Eq. (4.1) based on W and the tuple arrival rate r of the input stream monitored by the *Statistics Manager*.
3: Obtain estimations for the mean μ and the variance σ^2 of the values of the tuples in the most recent instantaneous window from the *Statistics Manager*. These estimations are based on tuples that have been received so far.
4: $Cvrg_{thr} \leftarrow$ the minimum $Cvrg$ that satisfies $(1 - Cvrg) + c_\delta \cdot \sqrt{(Cvrg - Cvrg^2) \cdot \frac{\sigma^2+\mu^2}{N \cdot \mu^2}} \leq \epsilon_{thr}$, i.e., Eq. (4.1), where c_δ is the critical value associated with the given confidence level δ according to the error function of a normal distribution.
5: **return** $Cvrg_{thr}$

The minimum value of $Cvrg$ that satisfies Eq. (4.1) is then the *window coverage threshold* $Cvrg_{thr}$ for the given error threshold (ϵ_{thr}, δ). The value of N in Eq. (4.1) can be calculated from the window size and the tuple arrival rate monitored by the *Statistics Manager* in Figure 4.1. The values of μ and σ can be estimated by the *Statistics Manager* as well.

The overall calculation of the window-coverage threshold $Cvrg_{thr}$ by the *Window-Coverage Threshold Calculator* for the SUM aggregate function is summarized in Algorithm 4.2. Leveraging results from the work of Law and Zaniolo [LZ08], $Cvrg_{thr}$ can be derived for many other aggregate functions including COUNT, AVG, QUANTILE, and complex aggregate functions that take these basic aggregates as building blocks.

The first calculation of $Cvrg_{thr}$ is at the end of the predefined warm-up period. To adapt to changes in the tuple arrival rate and the data characteristics within an input stream, in this dissertation, $Cvrg_{thr}$ is re-calculated periodically during the lifetime of a query (line 1 of Algorithm 4.2). However, more sophisticated strategies for re-calculating $Cvrg_{thr}$ can be applied as well; for example, the re-calculation can be triggered by changes detected in the stream statistics [KBG04; BG07].

Note that the calculation of $Cvrg_{thr}$ described above is based on the assumption that the random variables X_i are statistically independent. In sampling-based AQP, this assumption holds naturally because tuples are included into a sample in a random manner. In contrast, out-of-order tuples in a stream often present correlations. An out-of-order tuple $e_{i,j}$ is considered as a *correlated out-of-order tuple* if the tuple $e_{i,j-1}$ is an out-of-order tuple as well. The reason for such correlations between out-of-order tuples is that the out-of-order tuples are often caused by a faulty or overloaded data source, an unstable communication link, etc., which, once occurred, often would produce a sequence of out-of-order tuples. Indeed, strong correlations between out-of-order tuples have been observed in the real-world data streams that are used in the experimental evaluation of this dissertation (cf. Section 4.3.1). However, the exper-

imental results in Section 4.3.3 show that the above-described calculation of $Cvrg_{thr}$ works well for data streams with different disorder characteristics. The reasons are mainly twofold: (1) whether a tuple is an out-of-order tuple does not depend on the value of the tuple; (2) the aggregate result over Z does not depend on the order of the tuples within Z. Therefore, even if a sequence of timestamp-ordered tuples are delayed together and are absent in Z', conceptually, one could reorder the tuples in Z to randomize the positions of those missing tuples, and then define random variables X_i for the reordered Z. This reordering of Z does not influence the SUM result over Z.

In sampling-based AQP [LZ08; MZ10], each random variable X_i is defined to take the value $\frac{z_i}{p}$ with probability p and the value zero otherwise. As a result, the produced SUM result is the sum of the tuples in Z', scaled by the inverse of the sampling rate p. The goal is to compensate for the values of the sample-excluded tuples. In contrast, the *Window-Coverage Threshold Calculator* does not scale up the SUM result computed over Z' by the inverse of $Cvrg$ to compensate for the values of out-of-order tuples. The reason for this decision will be provided in Section 4.1.4.

4.1.4 Measuring Window Coverages at Runtime

The *Window-Coverage Runtime Calculator* in Figure 4.1 is responsible for measuring window coverages of instantaneous windows constructed after disorder handling. These window-coverage measurements are consumed by the *Buffer-Size Adaptation Unit* for different purposes when different buffer-size adaptation methods are applied. When the analytical-model-based adaptation method is applied, the measured window coverages are used to compute errors in the window coverages estimated by the analytical model. Based on these errors, the model-estimated window coverages are calibrated. When the PD-controller-based adaptation method is applied, the measured window coverages are indeed values of the *process variable* (cf. Section 3.4.2), which are input to the PD controller.

Denoting the total number of received tuples in an instantaneous window w as $N_{rcv}(w)$, and the number of unsuccessfully-handled out-of-order tuples for w, i.e., the number of missing tuples of w (cf. Section 3.4.1), as $N_{miss}(w)$, one can compute the window coverage of w as $Cvrg(w) = \frac{N_{rcv}(w)}{N_{rcv}(w)+N_{miss}(w)}$. The values of N_{miss} and N_{rcv} for each instantaneous window are maintained by the *Statistics Manager* in Figure 4.1. Specifically, the *Statistics Manager* uses a map $\mathcal{M}^w$ to maintain counters of missing tuples N_{miss} and counters of received tuple N_{rcv} for instantaneous windows that have been constructed. For each out-of-order tuple e that is handled unsuccessfully by the K-slack buffer in Figure 4.1, previously-constructed instantaneous windows, to which the tuple e belongs, are identified based on the timestamp of e and the window specification (W, β) (cf. Section 2.1.2). The N_{miss} counters of these instantaneous windows are then increased by one. For each in-order tuple and each successfully-handled out-of-order tuple, the instantaneous windows to which the tuple belongs are identified in the same way, and the corresponding N_{rcv} counters are increased by one.

Note that at the moment an instantaneous window is constructed at the SWA operator, e.g., the instantaneous window w in Figure 4.3a, the coverage of the instantaneous window measured in the above-described way is always 1. The reason is that

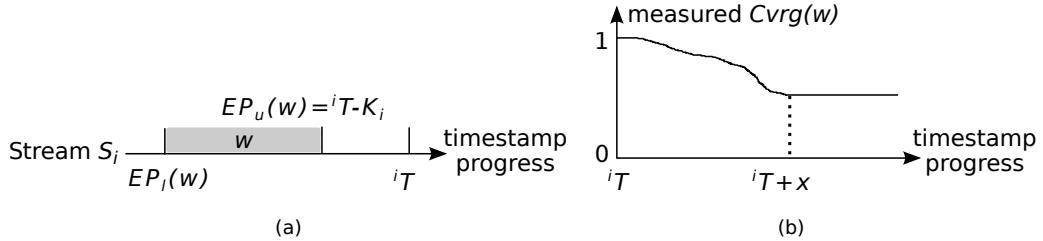


Figure 4.3: (a) The most recent instantaneous window constructed over an input stream under disorder handling; (b) The time-varying behavior of the measured coverage of the instantaneous window in the sub-figure (a).

at the moment an instantaneous window w is constructed, every previously-arrived out-of-order tuple either does not fall into the scope of w, and hence is irrelevant to the coverage of w; or falls into the scope of w, and has been sorted correctly by the K-slack buffer and included in w. Although in the input stream there might still exist out-of-order tuples that fall into the scope of w, these tuples can be detected only at later points in time. Only then, one can learn that the actual coverage of the instantaneous window w is smaller than 1. This time-varying behavior of the measured window coverage of an instantaneous window is depicted in Figure 4.3b. Starting from the moment an instantaneous window is constructed, its measured window coverage decreases gradually from 1 as more and more out-of-order tuples falling into the scope of the instantaneous window have been observed, and finally becomes stable when all out-of-order tuples falling into the scope of the instantaneous window have been received. Because the measured window coverage of the most recent instantaneous window is always 1, it is meaningless to scale up a SUM result by the inverse of the measured window coverage at the moment an instantaneous window is constructed.

Figure 4.3b suggests that the measured coverage of an instantaneous window gets more and more accurate over time. In other words, the measured window coverages of old instantaneous windows are more accurate than those of recent instantaneous windows. Obtaining accurate window-coverage measurements is especially important for the PD-controller-based buffer-size adaptation method; because otherwise it would make wrong adaptation decisions. In theory, without knowing the exact upper bound $MaxD_i$ of the tuple delays within an input stream S_i, one can never be certain about the point in time at which the coverage of an instantaneous window constructed over the stream becomes stable. Recall from Section 3.4.2 that, in this dissertation, $MaxD_i$ is estimated by the maximum delay among the so-far-arrived tuples in the streamS_i. Based on this estimation of $MaxD_i$, the coverage of an instantaneous window w can be assumed to be stable when the upper endpoint $EP_u(w)$ of w satisfies $EP_u(w) < {}^iT - MaxD_i$. In other words, the coverages of instantaneous windows constructed before ${}^iT - MaxD_i$ are assumed to be stable.

However, the disorder situation of a stream reflected in the measured window coverages of old instantaneous windows of the stream could be stale. On the one hand, it is desired to use accurate measurements of window coverages; on the other hand, it is also desired to make buffer-size adaptation decisions based on the fresh information of the stream disorder. Note that $MaxD_i$ represents the worst-case tuple delay, which normally occurs rarely. To reduce the effect of uncommon delay spikes

Algorithm 4.3 Measure the coverages of instantaneous windows constructed over a disorder-handled input stream S_i at the query runtime

Input: ${}^iT$ - the local current time at which a new instantaneous window is constructed
q - the retrospect parameter, $0 < q < 1$
Output: the window coverage *Cvrg* of a chosen instantaneous window
1: $M \leftarrow q$-quantile of the delays of out-of-order tuples observed so far
2: Among all instantaneous windows that were constructed before ${}^iT - M$, choose the instantaneous window w whose upper endpoint is closest to ${}^iT - M$.
3: Find the entry of w in the map $\mathcal{M}^w$ maintained by the *Statistics Manager*, and obtain the number of missing tuples N_{miss} and the number of received tuples N_{rcv} in w that have been observed by the time of ${}^iT$.
4: $Cvrg(w) \leftarrow \frac{N_{rcv}}{N_{rcv}+N_{miss}}$
5: **return** $Cvrg(w)$

on the freshness of the information used by the buffer-size adaptation method, and to take skews in the distribution of out-of-order tuples within the input stream into account, this dissertation proposes to measure the coverage of the instantaneous window that was constructed M time units earlier than ${}^iT$, where M is equal to the q-quantile ($0 < q < 1$) [Wan+13] of the delays of the so-far-arrived out-of-order tuples. The parameter q is referred to as the *retrospect parameter*, because it essentially determines how far to look back into the history of constructed instantaneous windows. q is a system parameter. It can be adjusted to trade off the accuracy of the window-coverage measurements against the freshness of the stream disorder situation reflected in the measurements.

Algorithm 4.3 summarizes the procedure of measuring the coverages of instantaneous windows constructed after disorder handling at the query runtime. To minimize the space consumption of the map $\mathcal{M}^w$ maintained by the *Statistics Manager*, $\mathcal{M}^w$ is purged periodically to remove entries of instantaneous windows that are very old, e.g., instantaneous windows that were constructed before ${}^iT - MaxD_i$.

4.1.5 Analytical-Model-Based Buffer-Size Adaptation

The *Buffer-Size Adaptation Unit* in Figure 4.1 is responsible for determining the updates of the K-slack buffer size, based on the information provided by the other components. This subsection describes the instantiation of the *Buffer-Size Adaptation Unit* using the analytical-model-based buffer-size adaptation method that was proposed in Section 3.4.1.

When using the analytical-model-based adaptation method, the coverage of the instantaneous window that will be constructed next is modeled as a function of the buffer size K, denoted by $Cvrg(w, K)$. In Section 3.4.1, it has been shown that the number of tuples included in the most recent instantaneous window constructed after disorder handling can be modeled as a function of K using Eq. (3.6). Based on the assumption that the near future resembles the recent past, the adaptation method estimates the number of tuples that would be included in the next instantaneous

Algorithm 4.4 Analytical-model-based adaptation of the K-slack buffer size to support QDDH for individual SWA queries

Input: W_i - window size on the input stream S_i
b - size of a basic window (cf. Section 3.4.1)
g - K-search granularity (cf. Section 3.4.1)
$Cvrg_{thr}$ - calculated window-coverage threshold (cf. Section 4.1.3)
Tuple-delay statistics from the *Statistics Manager* (cf. Section 3.4.1)
Output: κ - the K-slack buffer size to be applied before the next adaptation

1: $MaxD_i^R \leftarrow$ maximum tuple delay within the statistics window R_i^{stat} over the input stream S_i (cf. Section 3.4.1)
2: $\kappa \leftarrow 0$
3: **while** $(\kappa \leq MaxD_i^R \wedge Cvrg(w, \kappa) < Cvrg_{thr})$ **do**
4: $\quad \kappa \leftarrow \kappa + g$
5: **return** κ

window in the same way, and further estimates the coverage of that instantaneous window as

$$Cvrg(w, K) = \frac{|w|}{r \cdot W} \tag{4.2}$$

where $|w|$ is estimated by Eq. (3.6), r is the tuple arrival rate of the input stream, and W is the window size applied over the input stream. Note that the tuple arrival rate r appears in both the numerator (i.e., Eq. (3.6)) and the denominator of Eq. (4.2), and therefore can be canceled off.

With the relation between the window coverage and the buffer size K modeled by Eq. (4.2), the adaptation method then searches for the minimum possible value of K such that the condition $Cvrg(w, K) \geq Cvrg_{thr}$ is satisfied. This minimum possible value of K is referred to as the *optimal QDDH buffer size* of the SWA query, and is denoted by κ.

Algorithm 4.4 depicts the process of searching for the optimal QDDH buffer size κ in one iteration of the buffer-size adaptation. In this dissertation, κ is searched using a *trial and error* method. Specifically, let $MaxD_i^R$ denote the maximum tuple delay observed within the statistics window R_i^{stat} over the input stream (cf. Section 3.4.1). Algorithm 4.4 examines the κ values $\kappa = 0$, $\kappa = g$, $\kappa = 2g$, $\kappa = 3g$, etc. ($g > 0$), until either the examined κ value is greater than $MaxD_i^R$ or the condition $Cvrg(w, \kappa) \geq Cvrg_{thr}$ is satisfied. The last examined κ value is then returned as the optimal QDDH buffer size. Recall from Section 3.4.1 that the parameter g is called the *K-search granularity*, which defines the granularity in which the value domain of the delay of tuples is discretized to allow the tuple delay to be modeled by a discrete random variable.

To increase the accuracy of the window coverage estimated by Eq. (4.2), each estimated $Cvrg(w, \kappa)$ in Algorithm 4.4 can be calibrated based on the estimation errors in the past. The estimation error of an instantaneous window w can be obtained by comparing the last $Cvrg(w, \kappa)$ estimated for w with the $Cvrg$ value measured by the *Window-Coverage Runtime Calculator* for the same w.

4.1.6 Control-Based Buffer-Size Adaptation

When instantiating the *Buffer-Size Adaptation Unit* in Figure 4.1 using the PD-controller-based adaptation method introduced in Section 3.4.2, each time the *Buffer-Size Adaptation Unit* is invoked, the PD controller (cf. Figure 3.7) takes the window-coverage threshold $Cvrg_{thr}$ from the *Window-Coverage Threshold Calculator* as the setpoint, and the window-coverage measurement *Cvrg* from the *Window-Coverage Runtime Calculator* as the value of the process variable. Recall from Section 3.4.2 that the PD controller adjusts the parameter α, which is applied on top of $MaxD_i$. The optimal QDDH buffer size returned by the *Buffer-Size Adaptation Unit* at the end of an adaptation iteration is $\kappa = \alpha \cdot MaxD_i$.

4.2 QDDH for M-way Sliding-Window Join Queries

MSWJ queries are used in many stream-based applications for discovering correlations across different streams, e.g., finding similar news items from different news sources [Ged+07]. This section describes the instantiation of the generic QDDH framework introduced in Chapter 3 for individual MSWJ queries.

Formally, an MSWJ query has m ($m \geq 2$) input streams S_1, S_2, ..., S_m, and an optional join condition $p^{\bowtie}$, which may consist of one or more join predicates. Each input stream S_i is associated with a time-based sliding window, whose size W_i is specified by the user. Different input streams could have different window sizes. Semantically, an input tuple e_i from any input stream S_i joins with the subset of tuples $\{e_j | e_i.ts - W_j \leq e_j.ts \leq e_i.ts + W_i\}$ in every other stream S_j ($j \neq i$). A result tuple $\langle e_1, e_2, \ldots, e_m \rangle$ is produced if the join condition $p^{\bowtie}$ is met. The timestamp assigned to the result tuple is $max\{e_i.ts | i \in [1, m]\}$. Finding the optimal join order is orthogonal to disorder handling, and any existing work in this area (e.g., [VNB03; Bab+04]) can be applied.

If the disorder within the input streams is handled completely, then the join output, in terms of both the set of the result tuples and the order between the result tuples with respect to their timestamps[1], would be the same as the join output produced when the input streams do not have disorder at all. The number of result tuples produced at the arrival of a tuple e in the case that the stream disorder is absent is referred to as the *productivity* of the tuple e.

4.2.1 Result-Quality Metric

It has been shown in Example 3.1.2 that, when the disorder handling is incomplete, only a fraction of the true join results (i.e., results that would be produced if the input streams did not contain disorder) will be produced. The fraction of the actually-produced join results is defined as the *recall* [SW04b] of the join results, and is used as the result-quality metric for MSWJ queries.

In this dissertation, each time the recall of the join results needs to be measured, the join results whose timestamps are within the last P_{meas} time units, rather than all join results produced so far, are considered. The parameter P_{meas} is called the *result-quality measurement period*, and is a user-specified requirement. The parameter P_{meas}

[1] In this dissertation, the order of the result tuples that have the same timestamp is not restricted.

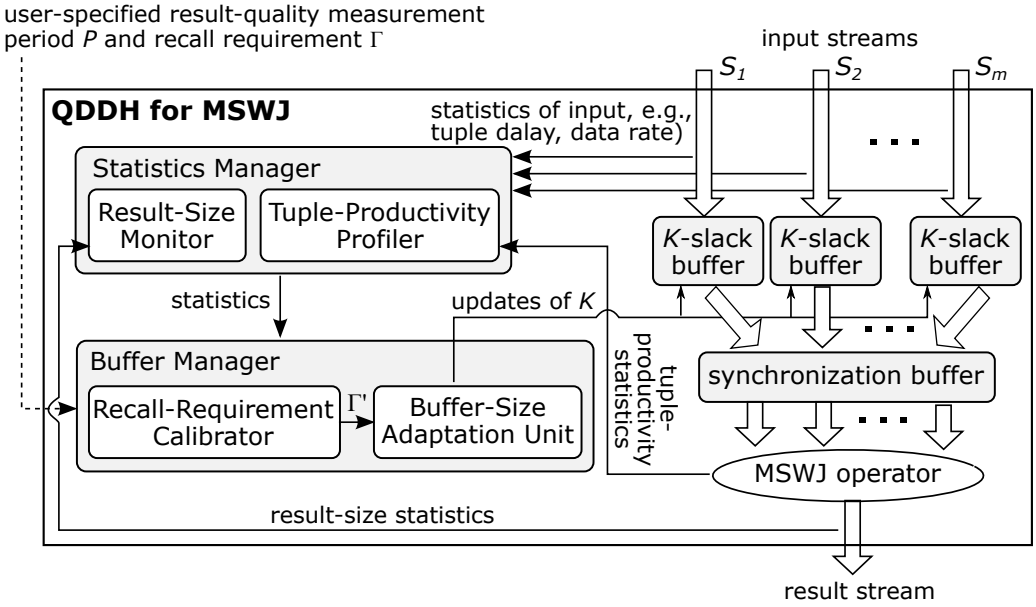


Figure 4.4: Instantiation of the buffer-based QDDH framework for individual MSWJ queries.

is introduced for two reasons: (1) it allows a user to specify the quality measurement period that is of his own interest; (2) with a full-history-based recall definition, it could happen that the fraction of the produced results is very high within a long period I_1, and is very low within the following long period I_2, but the overall fraction of the produced results within the period $I_1 + I_2$ still looks good. This may be undesirable for applications that would like to have continuously-high query-result quality. With a period-based recall definition, the situation described above is detectable if the length of the period is set small. In this regard, a period-based recall is indeed a stricter result-quality metric compared to a full-history-based recall. Using the period-based recall definition, the disorder handling procedure can be guided to provide a continuously-high query-result quality.

Formally, given a user-specified result-quality measurement period P_{meas}, the recall of the results of an MSWJ query measured at any time with respect to P_{meas} is denoted by $\gamma(P_{meas})$, and is defined as

$$\gamma(P_{meas}) = \frac{\text{\# join results whose timestamps are within the last } P_{meas} \text{ time units}}{\text{\# true join results whose timestamps are within the last } P_{meas} \text{ time units}}. \tag{4.3}$$

A user specifies the result-quality requirement for an MSWJ query by defining the minimum required $\gamma(P_{meas})$, which is denoted by Γ.

4.2.2 QDDH-Framework Instantiation Overview

Figure 4.4 depicts the instantiation of the generic QDDH framework for individual MSWJ queries. For the moment, let us assume an *MJoin*-style [VNB03] join implementation, where the join operation is conducted by a single join operator which takes m input streams directly, rather than by a tree of binary join operators [GÖ03b]. Same as for SWA queries, let us consider a physical MSWJ operator implementation that

implements both the logical window operators and the logical join operator. Dealing with the binary-tree-style join implementation will be discussed in Section 4.2.6.

Following the two-step disorder handling strategy described in Section 3.3, for each input stream of an MSWJ query, a K-slack buffer is used to handle the intra-stream disorder within the stream. The output streams of all K-slack buffers are then forwarded to a synchronization buffer, which handles the inter-stream disorder (cf. Algorithm 3.1).

To achieve the objective of QDDH, the *Buffer Manager* in Figure 4.4 adapts the K-slack buffer sizes every L_{adt} time units. L_{adt} is a system parameter and is referred to as the *adaptation interval*. The value of L_{adt} is set in such a way that it does not exceed the user-specified result-quality measurement period P_{meas}, i.e., $L_{adt} \leq P_{meas}$. At the end of each adaptation interval, the buffer size K of each K-slack buffer is determined for the next L_{adt} time units, based on the information provided by the *Statistics Manager*.

In addition to the statistics of the input streams, e.g., tuple delays and tuple arrival rates, the *Statistics Manager* has two sub-components: the *Tuple-Productivity Profiler* and the *Result-Size Monitor*. The *Tuple-Productivity Profiler* interacts with the join operator to monitor the productivity of an input tuple. The objective is to learn the potential correlation between the delay and the productivity of input tuples, which plays an important role in the analytical-model-based buffer-size adaptation method (cf. Section 4.2.4). The *Result-Size Monitor* maintains a sliding window of $P_{meas} - L_{adt}$ time units on the result stream. Based on the produced result size, the *Recall-Requirement Calibrator*—a sub-component of the *Buffer-Manager*—calibrates the user-specified recall requirement Γ to obtain a recall requirement to be used in a single adaptation iteration. This calibrated recall requirement is referred to as the *instant recall requirement*, denoted by Γ'. The calibration is based on the following intuition: If the recall of the join results produced within the last $P_{meas} - L_{adt}$ time units is much higher than the user-specified recall requirement Γ, then a lower recall requirement can be set for the join results produced within the next adaptation interval, which implies that smaller K-slack buffers can be used in the next adaptation interval; and vice versa. Same as in the instantiation of the QDDH-framework for individual SWA queries, new K-slack buffer sizes are determined by the *Buffer-Size Adaptation Unit* of the *Buffer Manager*, using a certain buffer-size adaptation method (cf. Section 4.2.4 and Section 4.2.5). The overall behavior of the *Buffer Manager* is summarized by Algorithm 4.5.

The output streams of the synchronization buffer are processed by the MSWJ operator, whose basic idea is described in Algorithm 4.6. Because of the lines 9–10 in Algorithm 3.1, the streams arriving at the join operator may still contain out-of-order tuples. The join operator can detect these out-of-order tuples by using a variable $^{\bowtie}T$ to track the maximum timestamp among the so-far-received tuples. A received tuple e_i ($i \in [1, m]$) is out of order if $e_i.ts < {}^{\bowtie}T$. For each received tuple e_i, if e_i is an in-order tuple, then $^{\bowtie}T$ is updated if $e_i.ts > {}^{\bowtie}T$, and e_i is processed following a three-step procedure: (1) Invalidate expired tuples in windows on all other streams (lines 5–6). (2) Join the tuple e_i with the remaining tuples in all other windows, and produce result tuples based on the given join condition (line 7). The timestamp assigned to each result tuple is $e_i.ts$. (3) Insert the tuple e_i into the window on the stream S_i (line 8). If the received tuple e_i is an out-of-order tuple, then step (1) and step (2)

Algorithm 4.5 The behavior of the *Buffer Manager* in the QDDH-framework instantiation for individual MSWJ queries (cf. Figure 4.4)

Input: specification of an MSWJ query
P_{meas} - user-specified result-quality measurement period
Γ - user-specified recall requirement on produced join results
L_{adt} - interval of adapting the K-slack buffer sizes for an MSWJ query statistics from the *Statistics Manager*
Output: new settings of the K-slack buffer sizes
1: **for each** adaptation interval L_{adt}, at the end of the interval **do**
2: Invoke the *Recall-Requirement Calibrator* to compute a calibrated recall requirement Γ'.
from the user-specified recall requirement Γ
3: Invoke the *Buffer-Size Adaptation Unit* with Γ' to determine the new K-slack buffer sizes to be applied during the next adaptation interval.

Algorithm 4.6 Execution of MSWJ over disorder-handled input streams

1: ${}^{\bowtie}T \leftarrow 0$
2: **for** each tuple e_i ($i \in [1, m]$) arrived at the join operator **do**
3: **if** $e_i.ts \geq {}^{\bowtie}T$ **then**
4: ${}^{\bowtie}T \leftarrow e_i.ts$
5: **for** the window on each stream S_j ($j \neq i$) **do**
6: Remove tuples e_j satisfying $e_j.ts < e_i.ts - W_j$ from the window
7: Probe the window on each stream S_j ($j \neq i$) and produce result tuples based on the join condition $p^{\bowtie}$
8: Insert e_i into the window on S_i
9: **else if** $e_i.ts > {}^{\bowtie}T - W_i$ **then**
10: Insert e_i into the window on S_i
11: Invoke the *Tuple-Productivity Profiler* to record the (estimated) productivity of e_i

are skipped; hence, (a fraction of) result tuples that can be derived from the tuple out-of-order e_i are lost. However, if the tuple e_i still falls into the current scope of the window on S_i (i.e., $e_i.ts \geq {}^{\bowtie}T - W_i$), then e_i could still contribute to deriving future result tuples. Hence, in this case, e_i is still inserted into the window on S_i (lines 9–10). Finally, the join operator invokes the *Tuple-Productivity Profiler* in Figure 4.4 to record the productivity of e_i. The productivity of an out-of-order tuple is estimated based on the join results produced in the past, which will be described in Section 4.2.4.

4.2.3 The *Same-K* Policy

Before describing the buffer-size adaptation performed by the *Buffer-Manager* in Figure 4.4 in detail, this subsection introduces one general policy that the *Buffer-Manager* follows in each iteration of the buffer-size adaptation. That is, all K-slack buffers applied for an MSWJ query use the same setting of K. This policy is termed as the *Same-K* policy, which is asserted by Theorem 4.1.

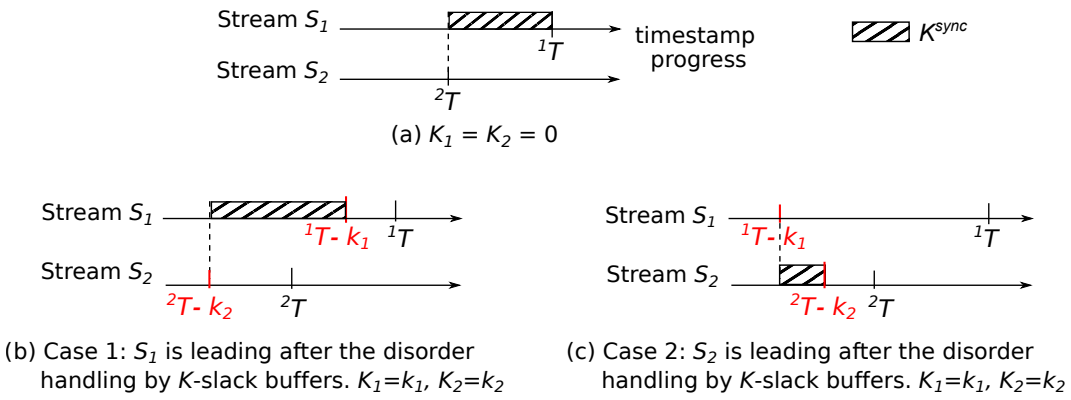


Figure 4.5: Illustrative proof of the *Same-K* policy for 2-way sliding-window joins.

Theorem 4.1. *For a query with multiple input streams $S_1, S_2, \ldots, S_m$, when using a two-step, buffer-based, prior-to-operation disorder handling strategy (cf. Section 3.3), then for any buffer-size configuration of the intra-stream disorder handling components (i.e., the K-slack buffers), where $K_1 = k_1$, $K_2 = k_2$, $\ldots$, $K_m = k_m$ (k_i is a constant and $k_i \geq 0$), the effect of the disorder handling under this configuration, and thus the produced join output, is the same as that under the configuration $K_1 = K_2 = \cdots = K_m = k$, where $k = min\{^iT | i \in [1,m]\} - min\{^iT - k_i | i \in [1,m]\}$.*

Basically, Theorem 4.1 says that, independent of the intra-stream disorder characteristics within the input streams of an MSWJ query, one could always find a configuration C where all K-slack buffers in Figure 4.4 apply the same setting of K, to replace another configuration C' where the K-slack buffers apply different settings of K, such that the join output produced under the configuration C is the same as the join output produced under the configuration C'. Hence, it suffices to use the same setting of K for all K-slack buffers.

Example 4.2.1. Consider a 2-way join with input streams S_1 and S_2, whose progress in terms of the tuple timestamp is as shown in Figure 4.5a: The stream S_1 is leading whereas the stream S_2 is lagging. When using a synchronization buffer to handle the inter-stream disorder within the two streams directly, the size of the synchronization buffer would be $^1T - {}^2T$. Recall from Section 3.2.2 that, even if the intra-stream disorder within each input stream is not handled by a K-slack buffer, i.e., $K_1 = K_2 = 0$, the synchronization buffer would handle it, at least partially, for the leading stream. Hence, for the case in Figure 4.5a, the buffer sizes for handling the inter-stream disorder in streams S_1 and S_2 are $^1T - {}^2T$ and 0, respectively.

Assume that the K-slack buffers applied for the streams S_1 and S_2 are configured as $K_1 = k_1$ and $K_2 = k_2$, and at least one of k_1 and k_2 is greater than zero. Then there are two possible cases[2]:

- *Case 1: The stream S_1 remains leading after the disorder handling by K-slack buffers (Figure 4.5b).* For the stream S_2, the total buffer size for handling its intra-stream disorder is k_2. For the stream S_1, in addition to the K-slack buffer, the

[2]The case in which both streams have the same timestamp progress after the disorder handling by K-slack buffers can be viewed as a special instance of either *Case 1* or *Case 2*.

synchronization buffer can further handle its intra-stream disorder. Th synchronization buffer size that contributes to handling the intra-stream disorder in S_1 is $K_1^{sync} = ({}^1T - k_1) - ({}^2T - k_2)$. Hence, the total buffer size for handling the intra-stream disorder in S_1 is $k_1 + K_1^{sync} = {}^1T - {}^2T + k_2$. Compared to the case in Figure 4.5a, for both streams, the total buffer size for handling the intra-stream disorder is increased by k_2, which is equivalent to having a K-slack buffer-size configuration where $K_1 = K_2 = k_2$.

- *Case 2: The stream S_2 becomes leading after the disorder handling by K-slack buffers (Figure 4.5c).* In this case, the synchronization buffer keeps and sorts tuples from the stream S_2, and $K_1^{sync} = 0$, $K_2^{sync} = ({}^2T - k_2) - ({}^1T - k_1)$. The total buffer size for handling the intra-stream disorder is k_1 for the stream S_1, and is $k_2 + K_2^{sync} = {}^2T - {}^1T + k_1$ for the stream S_2. Let $k = {}^2T - {}^1T + k_1$. Compared to the case in Figure 4.5a, the total buffer size for handling the intra-stream disorder is increased by k for both streams, which is equivalent to having a K-slack buffer-size configuration where $K_1 = K_2 = k$.

The formal proof of Theorem 4.1 is as follows.

Proof. The nature of Algorithm 3.1 determines that, for each synchronization buffer, the value of the variable T^{sync} is always determined by the maximum timestamp among the tuples of the slowest stream—in terms of the timestamp progress—that is input to the synchronization buffer. If the K-slack buffer applied over each input stream of the synchronization buffer is configured with a size of zero, then T^{sync} can be determined as $T^{sync} = min\{{}^iT | i \in [1, m]\}$, and K_i^{sync} for each input stream S_i can be determined as $K_i^{sync} = {}^iT - T^{sync} = {}^iT - min\{{}^iT | i \in [1, m]\}$. K_i^{sync} is also the total buffer size for handling the intra-stream disorder of the stream S_i under this buffer-size configuration. If the configuration for the K-slack buffer sizes is $K_1 = k_1$, $K_2 = k_2$, ..., $K_m = k_m$ ($k_i \geq 0$), then $T^{sync} = min\{{}^iT - k_i | i \in [1, m]\}$, and $K_i^{sync} = ({}^iT - k_i) - T^{sync} = {}^iT - k_i - min\{{}^iT - k_i | i \in [1, m]\}$. Now the total buffer size for handling the intra-stream disorder of the stream S_i is $k_i + K_i^{sync} = {}^iT - min\{{}^iT - k_i | i \in [1, m]\}$. Compared to the case where $K_i = 0$ for each $i \in [1, m]$, for each input stream S_i, the total buffer size for handling its intra-stream disorder is increased by $({}^iT - min\{{}^iT - k_i | i \in [1, m]\}) - ({}^iT - min\{{}^iT | i \in [1, m]\}) = min\{{}^iT | i \in [1, m]\} - min\{{}^iT - k_i | i \in [1, m]\}$. Hence, it is equivalent to having a K-slack buffer-size configuration where $K_1 = K_2 = \cdots = K_m = min\{{}^iT | i \in [1, m]\} - min\{{}^iT - k_i | i \in [1, m]\}$. □

The *Same-K* policy has another important benefit:

Proposition 4.1. *With the Same-K policy, for any two input streams S_i and S_j of an MSWJ query, the time skew (cf. Section 2.2.1) between their corresponding K-slack output streams is the same as the time skew between the streams S_i and S_j.*

Proof. $\forall i, j \in [1, m], \forall k, |({}^iT - k) - ({}^jT - k)| = |{}^iT - {}^jT|$. □

From the discussion above, it can be observed that K_i^{sync} for any stream is essentially the time skew between the stream's corresponding K-slack output stream and the slowest output stream among all K-slack buffers. Proposition 4.1 suggests that, with the *Same-K* policy, all K_i^{sync} can be determined directly based on the time skews

Algorithm 4.7 Analytical-model-based adaptation of the K-slack buffer sizes to support QDDH for individual MSWJ queries

Input: $\{W_i | i \in [1, m]\}$ - window sizes on input streams $S_1, S_2, \ldots, S_m$
b - size of a basic window (cf. Section 3.4.1)
g - K-search granularity (cf. Section 3.4.1)
L_{adt} - adaptation interval (cf. Section 4.2.2)
Γ' - instant recall requirement derived by *Recall-Requirement Calibrator*
Statistics maintained by the *Statistics Manager* (e.g., tuple delay, tuple productivity, and result size)
Output: κ - the K-slack buffer size to be applied in the next adaptation interval

```
1: MaxD^R ← max{MaxD_i^R | i ∈ [1, m]}
2: κ ← 0
3: while (κ ≤ MaxD^R ∧ γ(L_adt, κ) < Γ') do
4:     κ ← κ + g
5: return κ
```

between the raw input streams, regardless of the specific size settings for the K-slack buffers. The *Statistics Manager* in Figure 4.4 is responsible for monitoring K_i^{sync} at the query runtime.

4.2.4 Analytical-Model-Based Buffer-Size Adaptation

This subsection describes the instantiation of the *Buffer-Size Adaptation Unit* with the analytical-model-based adaptation method. Thanks to the *Same-K* policy introduced in Section 4.2.3, at each iteration of the buffer-size adaptation, the *Buffer-Size Adaptation Unit* only needs to determine a common K value for all the K-slack buffers in Figure 4.4. The objective is to find the minimum possible value of K to meet the user-specified recall requirement.

The analytical-model-based buffer-size adaptation method models the recall of the join results that would be produced in the next adaptation interval, whose length is L_{adt} (cf. Section 4.2.2), as a function of the buffer size K, denoted by $\gamma(L_{adt}, K)$. Algorithm 4.7 depicts the behavior of the method in a single iteration of the buffer-size adaptation. In each iteration, the method searches for the minimum possible value of K such that $\gamma(L_{adt}, K) \geq \Gamma'$ holds. This minimum possible value of K is then the optimal QDDH buffer size κ of the MSWJ query. The input Γ' of Algorithm 4.7 is an *instant recall requirement* derived by the *Recall-Requirement Calibrator* in Figure 4.4. It is derived for the recall measured over the period L_{adt}, from the user-specified requirement Γ for the recall measured over the result-quality measurement period P_{meas}. Details of this calibration procedure will be described later in this subsection.

Modeling $\gamma(L_{adt}, K)$

To estimate the recall of the join results produced within L_{adt} time units under a certain buffer-size configuration K, one needs to estimate the join result size that would be produced within L_{adt} under K, denoted by $N_{prod}^{\bowtie}(L_{adt}, K)$, and the true join result size within L_{adt}, denoted by $N_{true}^{\bowtie}(L_{adt})$, if the input streams did not contain disorder. $N_{true}^{\bowtie}(L_{adt})$ is independent of the configuration of K.

Estimating $N^{\bowtie}_{true}(L_{adt})$ For an MSWJ query with an arbitrary join condition $p^{\bowtie}$, $N^{\bowtie}_{true}(L_{adt})$ can be estimated by multiplying the result size of the corresponding *cross-join*, denoted by $N^{\times}_{true}(L_{adt})$, with the selectivity of the join condition, denoted by $sel^{\bowtie}$. The estimation of $sel^{\bowtie}$ will be discussed later. For the moment, let us assume that $sel^{\bowtie}$ is known. $N^{\times}_{true}(L_{adt})$ is the sum of the number of cross-join result tuples that would be produced at the arrival of each tuple e_i ($i \in [1,m]$) during the interval L_{adt} if the input streams did not contain disorder. The number of cross-join result tuples produced at the arrival of an individual tuple e_i from any input stream S_i is a simple product of the number of tuples within the most recent instantaneous window $w_{j,\top}$ on every other input stream S_j ($j \neq i$). The cardinality of $w_{j,\top}$, $|w_{j,\top}|$, can be estimated based on the average tuple arrival rate r_j of S_j and the window size W_j, i.e., $|w_{j,\top}| = r_j \cdot W_j$. For each stream S_i, the total number of tuples that would arrive during the interval L_{adt} can be estimated based on the average tuple arrival rate r_i as well, as $r_i \cdot L_{adt}$. In summary, $N^{\bowtie}_{true}(L_{adt})$ is estimated as

$$\begin{aligned} N^{\bowtie}_{true}(L_{adt}) &= sel^{\bowtie} \cdot N^{\times}_{true}(L_{adt}) = sel^{\bowtie} \cdot \sum_{i=1}^{m} (r_i \cdot L_{adt} \cdot \prod_{j=1, j\neq i}^{m} r_j \cdot W_j) \\ &= sel^{\bowtie} \cdot (\prod_{i=1}^{m} r_i) \cdot L_{adt} \cdot (\sum_{i=1}^{m} \prod_{j=1, j\neq i}^{m} W_j). \end{aligned} \tag{4.4}$$

Estimating $N^{\bowtie}_{prod}(L_{adt}, K)$ $N^{\bowtie}_{prod}(L_{adt}, K)$ is estimated again based on the result size of the corresponding cross-join under K, denoted by $N^{\times}_{prod}(L_{adt}, K)$, and the join selectivity under K, denoted by $sel^{\bowtie}(K)$. It can be observed that (1) the join operator produces result tuples, if any, only at the arrival of in-order tuples (cf. Algorithm 4.6); (2) when an in-order tuple e_i arrives at the join operator, the instantaneous window $w_{j,\top}$ ($j \neq i$) may be *incomplete*; because some tuples e_j that satisfy $e_j.ts \geq e_i.ts - W_j$ may have not arrived because of the incomplete disorder handling by the K-slack buffers and the synchronization buffer. Hence, to estimate $N^{\times}_{prod}(L_{adt}, K)$, one needs to estimate, under the given setting of K, the number of in-order tuples that the join operator would receive during L_{adt}, and the degree of completeness of the most recent instantaneous windows $w_{j,\top}$ ($j \neq i$) at the arrival of an in-order tuple e_i ($i \in [1,m]$). The number of in-order tuples that the join operator would receive during L_{adt} from the stream S_i can be estimated as $r_i \cdot L_{adt} \cdot f_{D^K_i}(0)$, where $f_{D^K_i}(0)$ represents the probability that the delay of a tuple in the corresponding, disorder-handled, derived stream of S_i is 0 (cf. Section 3.4.1). The cardinality of the instantaneous window $w_{j,\top}$ ($j \neq i$) can be estimated using Eq. (3.6). Overall, $N^{\bowtie}_{prod}(L_{adt}, K)$ can be estimated as

$$\begin{aligned} N^{\bowtie}_{prod}(L_{adt}, K) &= sel^{\bowtie}(K) \cdot \sum_{i=1}^{m} \left(r_i \cdot L_{adt} \cdot f_{D^K_i}(0) \cdot (\prod_{j=1, j\neq i}^{m} \sum_{l=1}^{n_j} |w^l_{j,\top}|) \right) \\ &= sel^{\bowtie}(K) \cdot (\prod_{i=1}^{m} r_i) \cdot L_{adt} \cdot \\ &\quad \sum_{i=1}^{m} \left(f_{D^K_i}(0) \prod_{j=1, j\neq i}^{m} \left(b \cdot \sum_{l=1}^{n_j - 1} \sum_{d=0}^{\frac{(l-1)b}{g}} f_{D^K_j}(d) + \left(W_j - (n_j - 1)b\right) \sum_{d=0}^{\frac{(n_j-1)b}{g}} f_{D^K_j}(d) \right) \right). \end{aligned} \tag{4.5}$$

Calculating $\gamma(L_{adt}, K)$ Having estimated $N^{\bowtie}_{true}(L_{adt})$ and $N^{\bowtie}_{prod}(L_{adt}, K)$, the recall $\gamma(L_{adt}, K)$ can be calculated as

$$\gamma(L_{adt}, K) = \frac{sel^{\bowtie}(K)}{sel^{\bowtie}} \cdot \frac{\sum_{i=1}^{m} \left(f_{D_i^K}(0) \prod_{\substack{j=1, \\ j \neq i}}^{m} \left(b \cdot \sum_{l=1}^{n_j - 1} \sum_{d=0}^{\frac{(l-1)b}{g}} f_{D_j^K}(d) + \left(W_j - (n_j - 1) \cdot b\right) \sum_{d=0}^{\frac{(n_j-1)b}{g}} f_{D_j^K}(d) \right) \right)}{\sum_{i=1}^{m} \prod_{j=1, j \neq i}^{m} W_j}, \quad (4.6)$$

where the common factor $(\prod_{i=1}^{m} r_i) \cdot L_{adt}$ in Eq. (4.4) and Eq. (4.5) is canceled off.

Learning Delay-Productivity Correlation and Estimating Join Selectivity

Eq. (4.6) requires estimating $\frac{sel^{\bowtie}(K)}{sel^{\bowtie}}$. A naive strategy for this estimation is to assume that the join selectivity under K, i.e., $sel^{\bowtie}(K)$, is equal to the true join selectivity $sel^{\bowtie}$. This is equivalent to estimating the recall $\gamma(L_{adt}, K)$ based on the result sizes of the corresponding cross-join. This strategy is denoted by *EqSel* hereafter. However, when stream disorder is present and the disorder handling is incomplete, the streams received by the join operator are different from the streams in the ideal case, where all input streams are in order and synchronized with each other. As a result, the join selectivity when stream disorder is present is often also different from the join selectivity in the ideal case. As an example, let us consider the 2-way join in Figure 4.6, where tuples are represented in the same way as in Figure 3.2; namely, the superscript of a tuple represents the timestamp of the tuple. If the input streams do not have disorder or if the disorder handling is complete, the join selectivity is $\frac{1}{3}$ (Figure 4.6a). If after disorder handling, a tuple in the stream S_1 arrives at the join operator out of order, then the join selectivity is no longer $\frac{1}{3}$ (Figure 4.6b and Figure 4.6c). Hence, it is more reasonable to assume that $sel^{\bowtie}(K)$ is different from $sel^{\bowtie}$. This strategy is denoted by *NonEqSel* hereafter.

To estimate $sel^{\bowtie}(K)$ for different configurations of K, one needs to consider the correlation between the delay and the productivity of tuples; because, as implied by Figure 4.6b and Figure 4.6c, tuples with different delays do not necessarily have the same productivity, and the unsuccessful handling of an out-of-order tuple that has a high productivity has a bigger influence on the produced recall than the unsuccessful handling of an out-of-order tuple that has a low productivity.

Extensive work exists, e.g., [SS94; RD08], which uses synopsis of input streams (e.g., histograms, sketches, and samples) to estimate the join result size or tuple productivities, and furthermore, the join selectivity. However, such *input-based* approaches do not work for joins with complex conditions, e.g., conditions involving user-defined functions [Cha09]. To support arbitrary join conditions and to be able to estimate $sel^{\bowtie}(K)$ for different configurations of K in each iteration of the buffer-size adaptation, this dissertation adopts an *output-based* approach; namely, the delay-productivity correlation is learned by monitoring the output of the join operator. Such

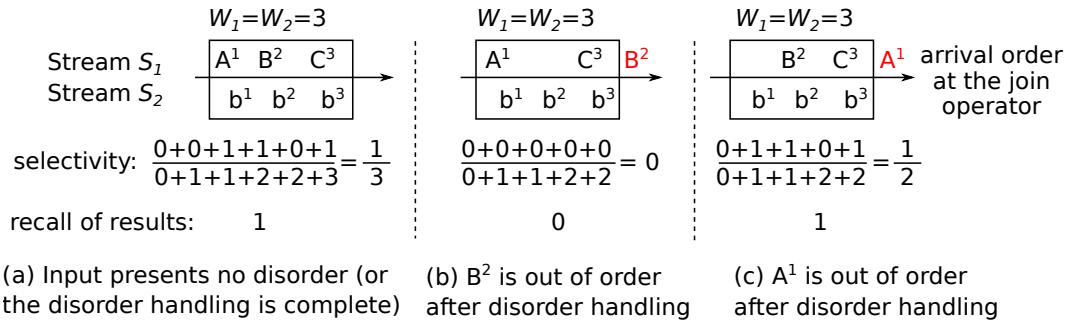


Figure 4.6: Effect of out-of-order tuples arriving at the join operator on the join selectivity and the recall of join results.

output-based approaches were also applied in prior work like [Bab+04; Ged+07; Lar+07] for different purposes.

Specifically, for each input tuple e_i ($i \in [1, m]$), the K-slack buffer applied for the stream S_i annotates e_i with its delay $delay(e_i)$. The tuple e_i carries this delay annotation through the synchronization buffer. If e_i arrives in order at the join operator, then during the join processing, the join operator records both the number of cross-join result tuples that e_i would derive, $n^{\times}(e_i)$, and the number of join result tuples that e_i actually derives, $n^{\bowtie}(e_i)$, given the content of the instantaneous windows $w_{j,\top}$ ($j \neq i$) over the other streams. The three numbers, $delay(e_i)$, $n^{\times}(e_i)$, and $n^{\bowtie}(e_i)$, are then provided to the *Tuple-Productivity Profiler* in Figure 4.4. The *Tuple-Productivity Profiler* uses two maps, $\mathcal{M}^{\times}$ and $\mathcal{M}^{\bowtie}$, to maintain the accumulated $n^{\times}$ and $n^{\bowtie}$, respectively, for each coarse-grained tuple delay observed within the last adaptation interval. The applied granularity for non-zero tuple delays is again g, which is consistent with the K-search granularity in Algorithm 4.7. If the tuple e_i arrives out of order at the join operator, then no join processing is conducted for e_i (cf. Algorithm 4.6). In this case, $n^{\bowtie}(e_i)$ and $n^{\times}(e_i)$ are estimated by the maximum $n^{\bowtie}(e)$ and $n^{\times}(e)$, respectively, over all in-order tuples e that have been received in the last adaptation interval.

Let $\mathcal{M}^{\times}[d]$ represent the value to which the coarse-grained tuple delay d maps in $\mathcal{M}^{\times}$, then $\mathcal{M}^{\times}[d] = \sum_{delay(e)=d} n^{\times}(e)$. Similarly, let $\mathcal{M}^{\bowtie}[d]$ represent the value to which the coarse-grained tuple delay d maps in $\mathcal{M}^{\bowtie}$, then $\mathcal{M}^{\bowtie}[d] = \sum_{delay(e)=d} n^{\bowtie}(e)$. Assuming that the join selectivity in the next adaptation interval is the same as the join selectivity in the last adaptation interval, for each K-slack buffer-size K examined in Algorithm 4.7, the join selectivity under K, $sel^{\bowtie}(K)$, is estimated as

$$sel^{\bowtie}(K) = \frac{\sum_{d=0}^{K} \mathcal{M}^{\bowtie}[d]}{\sum_{d=0}^{K} \mathcal{M}^{\times}[d]}.$$

The estimation of $sel^{\bowtie}$—the true join selectivity within L_{adt} if the input streams contained no disorder—is based on the following observation: For the join operator, the case where the input streams contain no disorder is equivalent to the case where the input streams contain disorder but the disorder is handled completely by using large-enough K-slack buffers. The join selectivity in the former case is the same as the join selectivity in the latter case. Hence, in this dissertation, the true join result size within the next adaptation interval is estimated based on the statistics collected in the last adaptation interval, as $\sum_{d=0}^{MaxD^{\mathcal{M}}} \mathcal{M}^{\bowtie}[d]$, where $MaxD^{\mathcal{M}}$ represents the current

maximum tuple delay in the map $\mathcal{M}^{\bowtie}$. The underlying rationale is that with a buffer of size $MaxD^{\mathcal{M}}$ time units, any tuple e whose delay satisfies $delay(e) \leq MaxD^{\mathcal{M}}$ can be re-ordered correctly by the buffer (cf. Section 3.2.1). Similarly, the result size of the corresponding cross join within the next adaptation interval is estimated as $\sum_{d=0}^{MaxD^{\mathcal{M}}} \mathcal{M}^{\times}[d]$. Now, the true join selectivity $sel^{\bowtie}$ can be estimated as

$$sel^{\bowtie} = \frac{\sum_{d=0}^{MaxD^{\mathcal{M}}} \mathcal{M}^{\bowtie}[d]}{\sum_{d=0}^{MaxD^{\mathcal{M}}} \mathcal{M}^{\times}[d]}. \tag{4.7}$$

In summary, the estimation for $\frac{sel^{\bowtie}(K)}{sel^{\bowtie}}$ is

$$\frac{sel^{\bowtie}(K)}{sel^{\bowtie}} = \frac{\sum_{d=0}^{K} \mathcal{M}^{\bowtie}[d]}{\sum_{d=0}^{K} \mathcal{M}^{\times}[d]} \cdot \frac{\sum_{d=0}^{MaxD^{\mathcal{M}}} \mathcal{M}^{\times}[d]}{\sum_{d=0}^{MaxD^{\mathcal{M}}} \mathcal{M}^{\bowtie}[d]}. \tag{4.8}$$

Deriving the Instant Recall Requirement

This part describes how the *instant recall requirement* Γ' needed in Algorithm4.7 is derived. This is done by using the runtime statistics maintained by the *Tuple-Productivity Profiler* and the *Result-Size Monitor* in Figure 4.4.

Given the user-specified result-quality measurement period P_{meas} (cf. Section 4.2.1), the *Result-Size Monitor* monitors the number of result tuples produced within the last $P_{meas} - L_{adt}$ time units, denoted by $N^{\bowtie}_{prod}(P_{meas} - L_{adt})$. Let $N^{\bowtie}_{true}(P_{meas} - L_{adt})$ denote the number of true join results that would be produced within the last $P_{meas} - L_{adt}$ time units if the input streams contained no disorder. In general, to make the recall measured at the end of the next adaptation interval for the results produced within the past P_{meas} time units meet the user-specified requirement Γ, the recall Γ' of the results produced within the next adaptation interval should satisfy Eq. (4.9).

$$\frac{N^{\bowtie}_{prod}(P_{meas} - L_{adt}) + N^{\bowtie}_{true}(L_{adt}) \cdot \Gamma'}{N^{\bowtie}_{true}(P_{meas} - L_{adt}) + N^{\bowtie}_{true}(L_{adt})} \geq \Gamma \tag{4.9}$$

Recall that the number of true join results within the next adaptation interval $N^{\bowtie}_{true}(L_{adt})$ can be estimated by $\sum_{d=0}^{MaxD^{\mathcal{M}}} \mathcal{M}^{\bowtie}[d]$, where $\mathcal{M}^{\bowtie}$ maintains the accumulated tuple productivities within the last adaptation interval. Furthermore, $N^{\bowtie}_{true}(P_{meas} - L_{adt})$ in Eq. (4.9) can be estimated by summing up the $N^{\bowtie}_{true}(L_{adt})$ estimations obtained in the last $(P_{meas} - L_{adt})/L_{adt}$ adaptation intervals. Together with $N^{\bowtie}_{prod}(P_{meas} - L_{adt})$ that is monitored by the *Result-Size Monitor*, the instant recall requirement Γ' can be derived from Eq. (4.9). The final instant recall requirement applied in Algorithm 4.7 is $max\{\Gamma', 1\}$.

4.2.5 Control-Based Buffer-Size Adaptation

When using the PD-controller-based buffer-size adaptation method to instantiate the *Buffer-Size Adaptation Unit* in the QDDH-framework instantiation for MSWJ queries (cf. Figure 4.4), the process variable maps to the recall $\gamma(P_{meas})$ of the join results produced within the last P_{meas} time units; and the setpoint maps to the user-specified recall requirement Γ. The output of the PD controller is again the adjustment of the

parameter α. However, in contrast to the case for a SWA query (cf. Section 4.1.6), where the parameter α is applied on top of the so-far-observed maximum tuple delay $MaxD_i$ within the only input stream of the SWA query, for an MSWJ query, the parameter α is applied on top of the so-far-observed maximum tuple delay across all input streams of the MSWJ query. Hence, the optimal QDDH buffer size returned by the *Buffer-Size Adaptation Unit* is $\kappa = \alpha \cdot max\{MaxD_i | i \in [1, m]\}$.

Compared with the case for SWA queries, it is more difficult for the PD controller to obtain values of the process variable (i.e., the recall of the produced join results) for MSWJ queries. The reason is that to compute the recall of the produced join results under a user-specified result-quality measurement period P_{meas}, the PD controller needs to know the true join result size within the last P_{meas} time units, which is determined by the values of tuples from multiple input streams as well as the join condition. Compared to the actual coverage of an individual instantaneous window, which is the process variable in the case for SWA queries, the true join result size is more difficult to obtain.

In this dissertation, the true join result size within the last P_{meas} time units is estimated in a similar way as in the procedure of deriving the instant recall requirement Γ' from Γ. Specifically, it is estimated as the sum of the $N^{\bowtie}_{true}(L_{adt})$ estimations obtained in the last P_{meas}/L_{adt} adaptation intervals. The produced join result size in the last P_{meas} time units can be obtained easily by the *Result-Size Monitor*, and the recall of the produced join results in the last P_{meas} time units can then be calculated.

4.2.6 Applicability in Distributed Join Processing

MSWJ queries are by nature CPU- and memory-intensive. To support a high volume of input tuples, large window sizes, and expensive join conditions, scalable and distributed processing of MSWJ queries has gained a lot of research interest recently (e.g., [WR09; Lin+15]). An MSWJ query can be implemented as either a single *MJoin*-style operator [VNB03] or a tree of binary join operators [GÖ03b]. Both types of implementation support distributed processing by splitting a macro m-way or binary join operator into smaller operator instances, exploiting the *pipelined parallelism* and the *data parallelism*.

As long as each operator instance follows the same processing semantics as depicted by Algorithm 4.6, then regardless of the specific type of the implementation, the instantiation of the QDDH framework described in this section can be adapted to be applied in a distributed setup in the following way: Same as in Figure 4.4, K-slack buffers are used to handle the intra-stream disorder of all input streams, and the *Buffer Manager* is responsible for adapting the K-slack buffer sizes at the query runtime. Each input stream to an operator instance in the distributed setup is either the output stream of a K-slack buffer, or the output stream of another operator instance[3]. To deal with the inter-stream disorder among the streams arriving at an operator instance, each operator instance is associated with a synchronization buffer. Indeed, such a prior-to-join stream-synchronization strategy has been applied in existing distributed join systems such as [WR09] and [Lin+15].

[3]The output of an operator instance is guaranteed to not contain intra-stream disorder because of the processing semantics of Algorithm 4.6.

With the analytical-model-based buffer-size adaptation method (cf. Section 4.2.4), the key information that is required by the *Buffer Manager* to make buffer-size adaptation decisions includes f_{D_i}, K_i^{sync}, $\mathcal{M}^{\times}$, $\mathcal{M}^{\bowtie}$, and $N_{true}^{\bowtie}(P_{meas} - L_{adt})$; all other information can be derived from the key information. Among the key information, f_D and K_i^{sync} can be obtained by monitoring the raw input streams of an MSWJ query, and $N_{true}^{\bowtie}(P_{meas} - L_{adt})$ can be obtained by monitoring the result stream that contains the final result tuples of the join. To build the map $\mathcal{M}^{\bowtie}$, each operator instance needs to be instrumented so that, when receiving a tuple e from a K-slack buffer, the operator instance annotates each intermediate result tuple produced at the arrival of e with the delay of e, $delay(e)$; and when receiving such an annotated, intermediate result tuple, the operator instance propagates the tuple-delay annotation further to each produced intermediate result tuple. Then, the map $\mathcal{M}^{\bowtie}$ can be built by monitoring the final join output. To build the map $\mathcal{M}^{\times}$ accurately, for each K-slack output tuple e_i ($i \in [1, m]$) that triggers the join processing at an operator instance, the *Buffer Manager* needs to know the cardinality $|w_{j,\top}|$ of the most recent instantaneous window $w_{j,\top}$ for each $j \in [1, m]$, $j \neq i$. However, each $w_{j,\top}$ is often split into slices, which are maintained by different operator instances. Hence, obtaining accurate $|w_{j,\top}|$ would require communicating with all involved operator instances, which can be expensive. An alternative is to approximate the window cardinality $|w_{j,\top}|$ using the average data rate r_j monitored by the *Statistics Manager* in Figure 4.4 and the window size W_j.

With the PD-controller-based buffer-size adaptation method (cf. Section 4.2.5), the *Buffer Manager* needs $\mathcal{M}^{\times}$, $\mathcal{M}^{\bowtie}$, and $N_{true}^{\bowtie}(P_{meas})$. $N_{true}^{\bowtie}(P_{meas})$ can again be obtained by monitoring the final result stream of an MSWJ query; and the maps $\mathcal{M}^{\times}$, $\mathcal{M}^{\bowtie}$ can be built in the same way as in the analytical-model-based buffer-size adaptation method described above.

4.3 Evaluation

This section presents the experimental evaluation of the two instantiations of the QDDH framework described in this chapter. For each instantiation, the evaluation aims to answer the following questions:

- Whether the objective of QDDH, i.e., quality-driven latency minimization, can be achieved with the proposed analytical-model-based and the control-based buffer-size adaptation methods, and how are the two adaptation methods compared to each other?
- How is the proposed QDDH approach compared with disorder handling approaches that make an extreme tradeoff between the end-to-end latency and the query-result quality?
- How does the configuration of important parameters involved in the two instantiations influence their performance in terms of achieving the objective of QDDH?
- What is the overhead of the quality-driven buffer-size adaptation in terms of the runtime?

	AggrDataset1	AggrDataset2
Duration (sec.)	980	1400
# tuples	544223	559211
# out-of-order tuples	313405	279337
Max. tuple delay (sec.)	4.49	16.95
# correlated out-of-order tuples	261426	231260

Table 4.1: General statistics of the real-world soccer-game data streams used in the evaluation of QDDH for individual SWA queries.

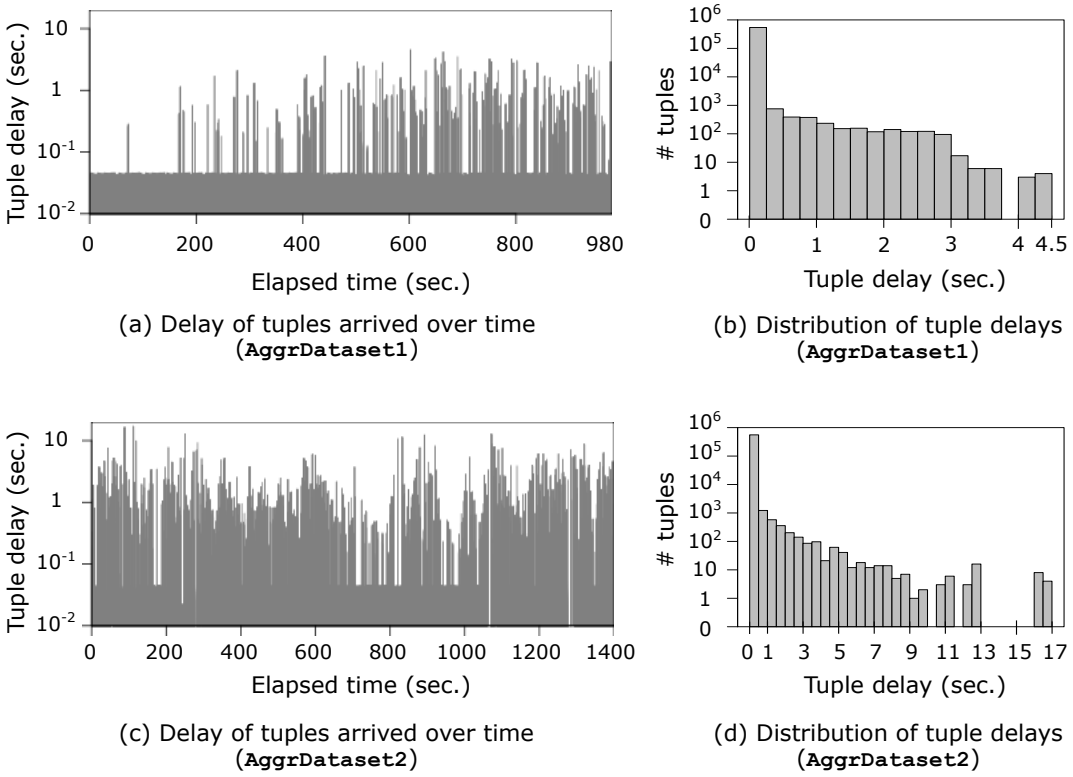
4.3.1 Implementation and Setup

All instantiations of the QDDH framework proposed in this dissertation were implemented in a prototypical version of SAP Event Stream Processor (SAP ESP) [SAP]. SAP ESP is a general-purpose DSPS that supports both conventional relational query operators like select, aggregate, join, etc., as well as pattern detection operators (cf. Section 2.1). All experiments in this section were conducted on a HP Z620 workstation, which has 24 cores (2.9GHz per core) and 96GB RAM, running SUSE 11.2.

Datasets and Queries

QDDH for SWA The evaluation of QDDH for individual SWA queries used two sensor-data streams, denoted by `AggrDataset1` and `AggrDataset2`. The two streams were produced by a Real-time Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany. The stream data was collected during soccer training games, tracking positions and velocities of soccer players [MZJ13]. Each original stream was projected onto the schema (`ts`, `vel`), where `ts` stands for the timestamp and `vel` stands for the velocity. Table 4.1 summarizes the general statistics of the two streams, and Figure 4.7 shows the disorder characteristics of the two streams. Both streams contain approximately 50% out-of-order tuples. The majority of the out-of-order tuples are delayed by less than 100 milliseconds. Recall from Section 4.1.3 that an out-of-order tuple $e_{i,j}$ in a stream S_i is considered to be a correlated out-of-order tuple if the tuple $e_{i,j-1}$ in the stream S_i is an out-of-order tuple as well. Strong correlations between out-of-order tuples were observed in both `AggrDataset1` and `AggrDataset2`. Compared to `AggrDataset1`, `AggrDataset2` contains more out-of-order tuples with large delays, and has a higher maximum tuple delay. In addition, in `AggrDataset1`, tuples with large delays appear mainly in the second half of the stream; whereas in `AggrDataset2`, tuples with large delays are distributed more or less uniformly within the entire stream.

For each stream, sliding-window SUM queries with varying window sizes were evaluated. The window slide β in all queries was set to 0.1 second. To compute the relative error of each produced aggregate result, sorted versions of `AggrDataset1` and `AggrDataset2` were generated. The exact results of each sliding-window SUM query used in the evaluation can then be obtained by evaluating the same query over the sorted versions of the two datasets.



(a) Delay of tuples arrived over time (**AggrDataset1**)

(b) Distribution of tuple delays (**AggrDataset1**)

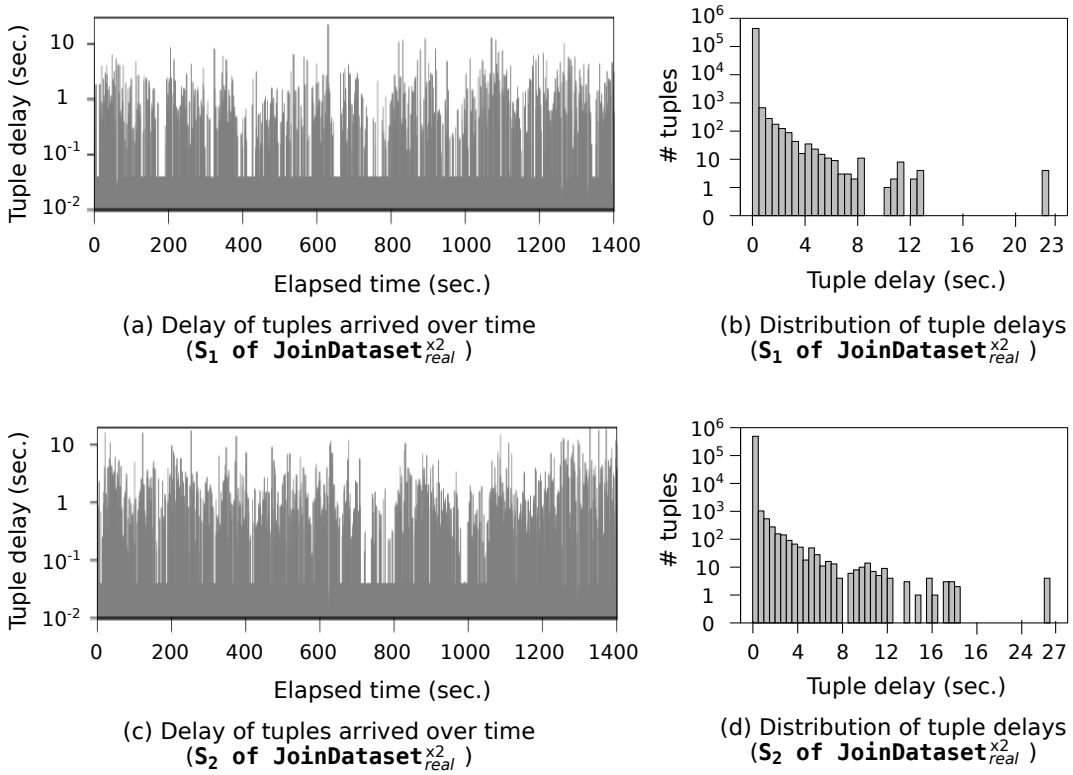
(c) Delay of tuples arrived over time (**AggrDataset2**)

(d) Distribution of tuple delays (**AggrDataset2**)

Figure 4.7: Disorder characteristics of the real-world soccer-game data streams used in the evaluation of QDDH for individual SWA queries.

QDDH for MSWJ To evaluate QDDH for individual MSWJ queries, one real-world dataset with two input streams and two synthetic datasets with three and four input streams were used. A different join query was used for each of the three datasets:

- The real-world dataset $\texttt{JoinDataset}^{\times 2}_{real}$ contains again soccer-game data, which was collected by the RTLS mentioned above. The original single stream from a 23-minute soccer game was split into two streams ($\texttt{S}_1$ and $\texttt{S}_2$), each containing the data of one team in the game. Moreover, the original stream was projected onto `(ts, sID, xCoord, yCoord)`, where `sID` identifies players and the pair of coordinates (`xCoord`, `yCoord`) encodes positions in the field. The streams $\texttt{S}_1$ and $\texttt{S}_2$ both contain approximately 450k tuples. The maximum tuple delay is 22 seconds in the stream $\texttt{S}_1$ and is 26.3 seconds in the stream $\texttt{S}_2$. Figure 4.8 plots the disorder characteristics of the two streams. The join query $Q^{\times 2}$ evaluated on $\texttt{JoinDataset}^{\times 2}_{real}$ is to find all occurrences, within a 5-second sliding window, where the distance between two players, one from each team, is smaller than 5 meters. A custom function `dist()` was used to calculate the distance and is the join condition in this scenario.
  $Q^{\times 2}$: `SELECT * FROM` $\texttt{S}_1$ `[5 SEC],` $\texttt{S}_2$ `[5 SEC]`
  `WHERE dist(`$\texttt{S}_1$`.xCoord,` $\texttt{S}_1$`.yCoord,` $\texttt{S}_2$`.xCoord,` $\texttt{S}_2$`.yCoord)` < 5
- The first synthetic dataset $\texttt{JoinDataset}^{\times 3}_{syn}$ consists of three streams, which have the same schema `(ts, a1)`. All streams start from a common timestamp ts^{init} and cover an interval of 30 minutes. The timestamps are in the granularity



(a) Delay of tuples arrived over time ($\mathbf{S_1}$ **of** $\mathtt{JoinDataset}^{\times 2}_{real}$)

(b) Distribution of tuple delays ($\mathbf{S_1}$ **of** $\mathtt{JoinDataset}^{\times 2}_{real}$)

(c) Delay of tuples arrived over time ($\mathbf{S_2}$ **of** $\mathtt{JoinDataset}^{\times 2}_{real}$)

(d) Distribution of tuple delays ($\mathbf{S_2}$ **of** $\mathtt{JoinDataset}^{\times 2}_{real}$)

Figure 4.8: Disorder characteristics of the real-world soccer-game data streams used in the evaluation of QDDH for individual MSWJ queries.

of millisecond (ms). For each stream $\mathtt{S_i}$ ($i \in \{1,2,3\}$) in $\mathtt{JoinDataset}^{\times 3}_{syn}$, tuples were generated sequentially as follows. Initially, let ${}^iT = ts^{init}$. For each new tuple e, the variable ${}^iT$ was increased by 10 ms (i.e., ${}^iT \mathrel{+}= 10$), and a random delay $delay(e)$ was chosen between (inclusive) 0.0 second and 20.0 seconds based on a Zipf distribution with skew zs_i^d. The timestamp of the new tuple e, $e.ts$, was then set to ${}^iT$ if $delay(e) = 0$, or to ${}^iT - delay(e)$ otherwise. Increasing ${}^iT$ by 10 ms for each newly-generated tuple simulates a data rate of 100 tuples per second. The applied Zipf skews zs_i^d for all three streams were $zs_1^d = 2.0$ and $zs_2^d = zs_3^d = 3.0$. The value of the attribute `a1` for a new tuple e was generated randomly from the integer interval $[1, 100]$, based on a Zipf distribution as well. To simulate a time-varying join selectivity, for each stream, the Zipf skew zs_i^{a1} for generating values of the attribute `a1` was initialized to 1.0, and was changed, within the range $[0.0, 5.0]$, during the data generation. The time interval between two consecutive changes of zs_i^{a1} was chosen randomly between (inclusive) 1 minute and 10 minutes. The three streams are synchronized with each other. A 3-way sliding-window join query $Q^{\times 3}$ was evaluated on $\mathtt{JoinDataset}^{\times 3}_{syn}$:

$Q^{\times 3}$:
```
SELECT * FROM S1 [5 SEC], S2 [5 SEC], S3 [5 SEC]
  WHERE S1.a1=S2.a1 AND S2.a1=S3.a1
```

- The second synthetic dataset $\mathtt{JoinDataset}^{\times 4}_{syn}$ consists of four streams, whose schemas are $\mathtt{S_1}$:(ts, a1, a2, a3), $\mathtt{S_2}$:(ts, a1), $\mathtt{S_3}$:(ts, a2), and $\mathtt{S_4}$:(ts,

a3). The timestamps and attribute values of the stream tuples were generated in the same way and from the same domains as for $\texttt{JoinDataset}_{syn}^{\times 3}$. The Zipf skew used for generating each attribute was also initialized to 1.0. The Zipf skews used for generating tuple delays were $zs_1^d = zs_2^d = zs_3^d = 3.0$ and $zs_4^d = 4.0$. A 4-way sliding-window join query $Q^{\times 4}$ was evaluated on $\texttt{JoinDataset}_{syn}^{\times 4}$:

$Q^{\times 4}$: `SELECT * FROM S1 [3 SEC], S2 [3 SEC], S3 [3 SEC], S4 [3 SEC]`
`WHERE S1.a1=S2.a1 AND S1.a2=S3.a2 AND S1.a3=S4.a3`

For each of the above three datasets, a sorted version where tuples of all streams in the dataset are ordered globally according to their timestamps was generated. By evaluating the query $Q^{\times x}$ ($x \in \{2, 3, 4\}$) over the corresponding sorted dataset, the true join results can be obtained, and the recall of the join results produced for the unsorted dataset can then be calculated therewith.

Performance Metrics

To evaluate the performance of the two proposed instantiations of the QDDH framework in terms of the achieved tradeoff between the end-to-end latency and the query-result quality, for each evaluated query, the following two metrics are considered:

- The average end-to-end latency.
- The overall query-result quality measured in terms of the *requirement fulfillment ratio* Φ, which is defined based on the query type as follows:
  For a SWA query, Φ is defined with respect to the user-specified result relative-error threshold ϵ_{thr} as

$$\Phi(\epsilon_{thr}) = \frac{\text{number of produced aggregate results that satisfy } \epsilon \leq \epsilon_{thr}}{\text{total number of true aggregate results}} \quad (4.10)$$

  For an MSWJ query, Φ is defined with respect to the user-specified recall requirement Γ as

$$\Phi(\Gamma) = \frac{\text{number of } \gamma(P_{meas}) \text{ measurements that satisfy } \gamma(P_{meas}) \geq \Gamma}{\text{total number of } \gamma(P_{meas}) \text{ measurements}} \quad (4.11)$$

  The recall $\gamma(P_{meas})$ of join results produced under disorder handling was measured right before each adaptation of the K-slack buffer size K. The recall measurements obtained during the first quality-measurement period P_{meas} were excluded when computing $\Phi(\Gamma)$.

Default Parameter Configuration

Unless otherwise stated, the experiments presented in this section used the parameter configurations as listed in Table 4.2. The parameter tuning for the family of PID controllers is a broad area in itself. In this dissertation, manual tuning based on the well-known *Ziegler-Nichols method* [ZN42] was applied to configure the parameters U_p and U_d of the PD controller used in the proposed control-based buffer-size adaptation method (cf. Section 3.4.2), so that they can minimize the consequent K-slack buffer sizes while respecting the user-specified result-quality requirements.

Parameter	Default value
basic window size b (common in both instantiations)	10 milliseconds
K-search granularity g (common in both instantiations)	10 milliseconds
U_p of the PD-controller in QDDH for SWA	0.2
U_d of the PD-controller in QDDH for SWA	4
retrospect parameter q in QDDH for SWA	0.99
confidence level δ of error threshold in QDDH for SWA	0.05
U_p of the PD-controller in QDDH for MSWJ	0.6
U_d of the PD-controller in QDDH for MSWJ	0.8
result-quality measurement period P_{meas} in QDDH for MSWJ	1 minute
buffer-size adaptation interval L_{adt} in QDDH for MSWJ	1 second

Table 4.2: Default parameter setting applied in the evaluation of the instantiations of the QDDH framework.

4.3.2 Baseline Disorder Handling Approaches and Results

For each instantiation of the QDDH framework introduced in this chapter, the proposed implementation of the respective *Buffer Manager* (cf. Figure 4.1 and Figure 4.4) was compared with two baseline implementations, which manage the sizes of the applied K-slack buffers in two extreme ways:

1. *No-K-slack*, which does not handle the intra-stream disorder of each input stream with a K-slack buffer at all, i.e., $K_i = 0$ for any input stream S_i. This approach produces the lowest end-to-end latency and the lowest query-result quality.

2. *Max-K-slack*, which updates the size K of each K-slack buffer dynamically to make K equal the maximum delay among the so-far-observed tuples from all involved input streams [MP13a]. This approach produces the highest end-to-end latency and the highest query-result quality.

Baseline Results of QDDH for SWA

Figure 4.9 plots the cumulative distribution function (CDF) of the relative error of the results produced by the *No-K-slack* baseline approach for five sliding-window SUM queries, whose window sizes range from 0.1 second to 10 seconds. The CDF curves do not intersect with the y-axis at 0 because the x-axis starts from 0.01%, not 0%. This figure shows how much the intra-stream disorder can degrade the accuracy of aggregate results if it is not handled at all. It can be observed that, in general, the stream disorder impairs the result accuracy more for queries with small window sizes than for queries with large window sizes. For instance, for the query whose window size is $W = 10$ seconds, the relative error of 1% or less was observed in 99% of the results produced from `AggrDataset1`, and in 97% of the results produced from `AggrDataset2`. Such a query-result quality may meet the requirements of most applications. However, as the window size decreases, the degradation of the result accuracy becomes more and more significant. For the query whose window size is $W = 0.1$ second, the relative error of 10% or more was observed in nearly half of

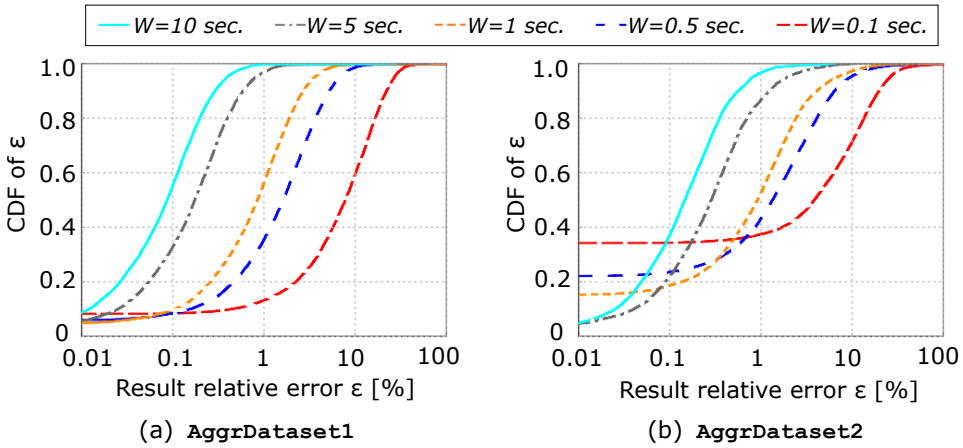


Figure 4.9: Cumulative distribution functions (CDF) of the relative errors of the aggregate results produced by the *No-K-slack* baseline approach for `AggrDataset1` and `AggrDataset2`.

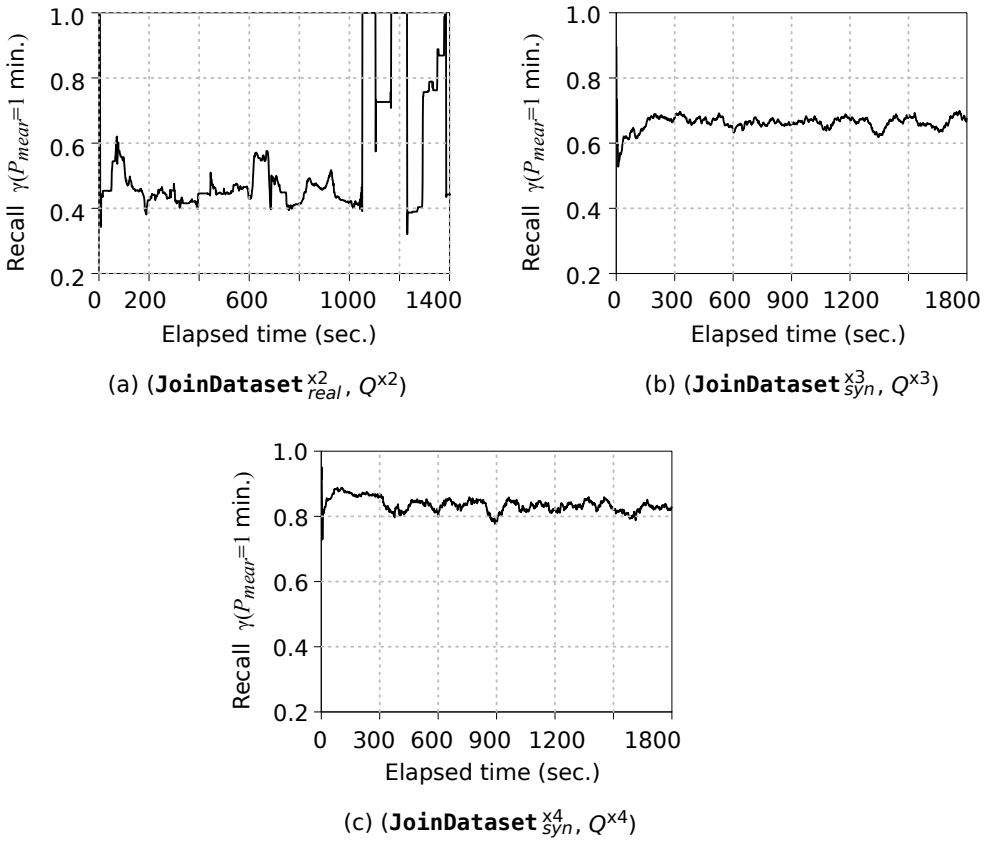
		Window size W (sec.)				
		0.1	0.5	1	5	10
# wrong query results	`AggrDataset1`	9	28	40	45	45
	`AggrDataset2`	12	37	57	100	12
Minimum non-zero result relative error ϵ [%]	`AggrDataset1`	2.16	0.31	0.24	0.05	0.05
	`AggrDataset2`	1.54	0.01	0.01	0.01	0.15
Maximum result relative error ϵ [%]	`AggrDataset1`	44.66	14.07	5.54	0.95	0.06
	`AggrDataset2`	28.27	28.27	100	0.53	28.27

Table 4.3: Accuracy of aggregate results produced by the *Max-K-slack* baseline approach for `AggrDataset1` and `AggrDataset2`. There are in total 9800 and 14000 results for `AggrDataset1` and `AggrDataset2`, respectively.

the results produced from both datasets, which suggests the necessity of disorder handling.

For each dataset, the average end-to-end latencies of all five queries were similar. For `AggrDataset1`, the average latency was around 0.22 second and for `AggrDataset2`, it was around 0.014 second.

Table 4.3 summarizes the accuracy of the query results produced by the *Max-K-slack* baseline approach. The average end-to-end latencies of all five queries were around 3.05 seconds for `AggrDataset1` and 17.3 seconds for `AggrDataset2`. Note that *Max-K-slack* does not guarantee complete disorder handling; because in *Max-K-slack*, each increase of the K-slack buffer size is caused by an out-of-order tuple that is not handled successfully by the buffer. These unsuccessfully-handled out-of-order tuples lead to wrong aggregate results. However, the total number of wrong results was very small. Even in the worst case, i.e., the query with $W = 5$ seconds executed over `AggrDataset2`, only about 0.7% ($\approx \frac{100}{14000}$) of the produced results were wrong.



(a) (**JoinDataset**$^{x2}_{real}$, Q^{x2})

(b) (**JoinDataset**$^{x3}_{syn}$, Q^{x3})

(c) (**JoinDataset**$^{x4}_{syn}$, Q^{x4})

Figure 4.10: Recall of the join results produced by the *No-K-slack* baseline approach for (JoinDataset$^{\times x}$, $Q^{\times i}$), $i \in \{2,3,4\}$.

Baseline Results of QDDH for MSWJ

Figure 4.10 shows the recall $\gamma(P_{meas})$ of the join results produced by the *No-K-slack* baseline approach for each JoinDataset$^{\times i}$ ($i \in \{2,3,4\}$). For (JoinDataset$^{\times 2}_{real}$, $Q^{\times 2}$), the measured recall was only around 0.5 for the most of the time. The overall recall for (JoinDataset$^{\times 4}_{syn}$, $Q^{\times 4}$) was the highest, but was only around 0.8, which is still a low result quality for many stream-based applications. Recall from Figure 4.4 that the synchronization buffer is always applied in the instantiation of the QDDH framework for MSWJ queries to handle the inter-stream disorder. Figure 4.10 implies that, to obtain a high result quality for MSWJ queries, handling only the inter-stream disorder is not sufficient, and the intra-stream disorder handling is necessary. The average end-to-end latency was 0.31 second for (JoinDataset$^{\times 2}_{real}$, $Q^{\times 2}$), 1.4 second for (JoinDataset$^{\times 3}_{syn}$, $Q^{\times 3}$), and 1.18 second for (JoinDataset$^{\times 4}_{syn}$, $Q^{\times 4}$).

Table 4.4 lists the average end-to-end latency and the average recall of the join results (i.e., the average over all $\gamma(P_{meas})$ measurements of a query) produced by the *Max-K-slack* baseline approach. Again, because *Max-K-slack* does not guarantee a complete disorder handling, the average $\gamma(P_{meas})$ was not always 1.

	Avg. latency (sec.)	Avg. $\gamma(P_{meas})$
$(\texttt{JoinDataset}^{\times 2}_{real}, Q^{\times 2})$	20.86	1.0
$(\texttt{JoinDataset}^{\times 3}_{syn}, Q^{\times 3})$	21.43	0.999
$(\texttt{JoinDataset}^{\times 4}_{syn}, Q^{\times 4})$	15.1	0.999

Table 4.4: Experimental results of the *Max-K-slack* baseline approach for $(\texttt{JoinDataset}^{\times x}, Q^{\times i})$, $i \in \{2, 3, 4\}$.

4.3.3 Effectiveness of QDDH

The experiments presented in this subsection aim to study the effectiveness the two proposed QDDH-framework instantiations in performing quality-driven latency minimization under varying settings of the respective result-quality requirement. In addition, the analytical-model-based buffer-size adaptation method is compared with the PD-controller-based adaptation method for each framework instantiation. For ease of reference, in the remainder of this evaluation section, the analytical-model-based adaptation method is denoted by *AM-adt*, and the PD-controller-based adaptation method is denoted by *PD-adt*.

Effectiveness Results of QDDH for SWA

Based on the experimental results of the *No-K-slack* baseline approach in Figure 4.9, the four sliding-window SUM queries whose window sizes are 0.1, 0.5, 1, and 5 seconds were chosen to evaluate the effectiveness of QDDH for individual SWA queries; because the result accuracy of these four queries has larger room for improvement than that of the query whose window size is $W = 10$ seconds. Each chosen query was evaluated over `AggrDataset1` and `AggrDataset2` using both the *AM-adt* (cf. Section 4.1.5) and the *PD-adt* (cf. Section 4.1.6) methods, under a series of ϵ_{thr} values ranging from 0.01% to 10%. The confidence level δ in each relative-error threshold took the default configuration (cf. Table 4.2).

The experimental results are shown in Figure 4.11 and Figure 4.12. For ease of comparison, the requirement fulfillment ratio produced by the *No-K-slack* baseline approach and the average end-to-end latency produced by the *Max-K-slack* baseline approach are included in the figures as well. In general, for both buffer-size adaptation methods, the average end-to-end latency decreases as ϵ_{thr} increases. This behavior is as expected: intuitively, a higher ϵ_{thr} value means that a higher number of missing tuples are allowed in each instantaneous window constructed over the input stream, and therefore a smaller K-slack buffer can be used. The inverse trend between the average latency and the applied ϵ_{thr} value also shows that the proposed QDDH approach can dynamically minimize the K-slack buffer size, thus the end-to-end latency, to achieve different levels of result accuracy specified for a SWA query. This inverse trend is more obvious for queries with large window sizes than for queries with small window sizes.

AM-adt and *PD-adt* achieved a requirement fulfillment ratio $\Phi(\epsilon_{thr})$ of at least 92% and 95%, respectively. However, in the majority of the test cases, *AM-adt* produced a smaller end-to-end latency compared to *PD-adt*. The reason is that at each iteration of the buffer-size adaptation, *AM-adt* attempts to find the optimal buffer size to be

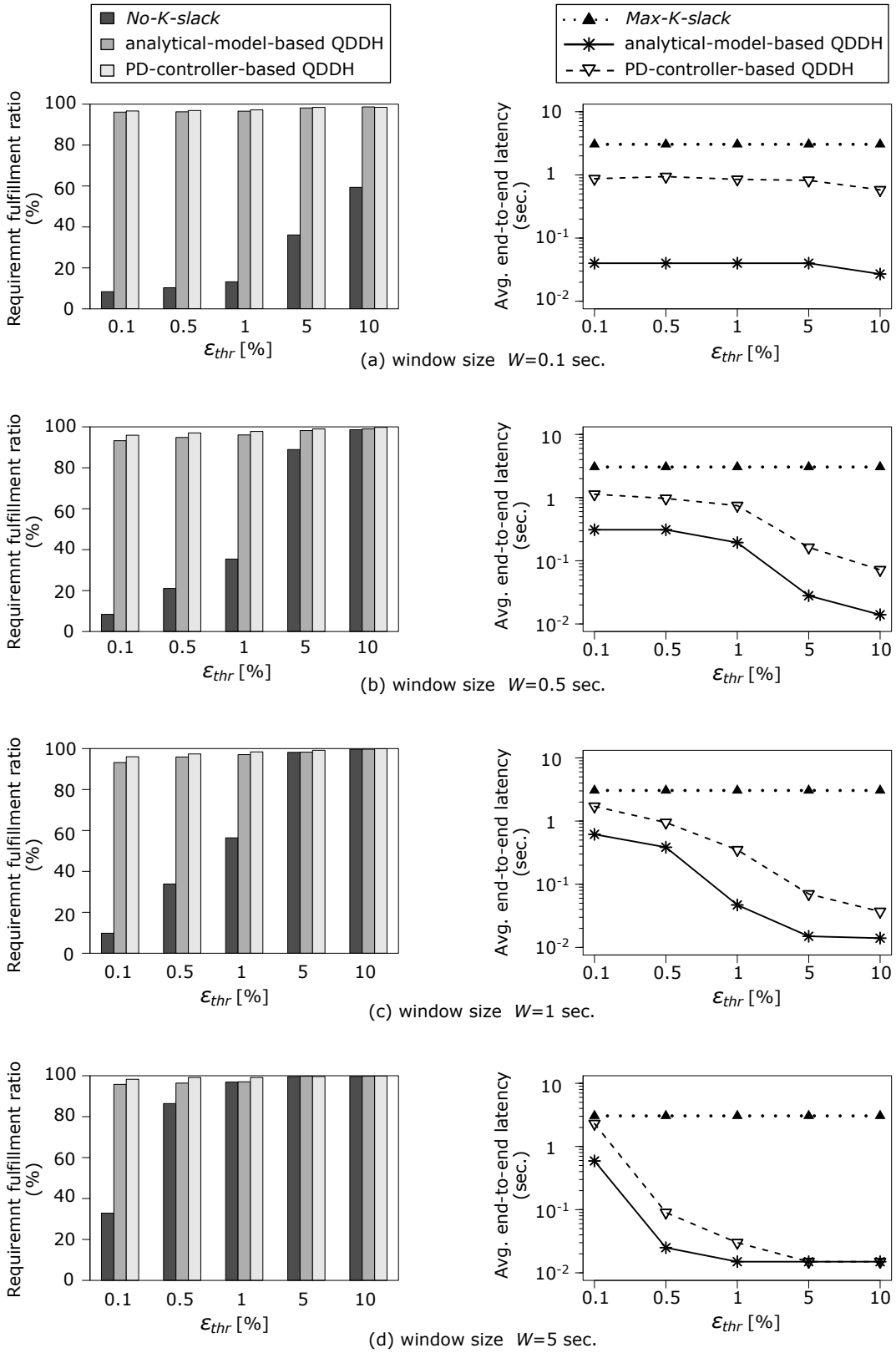


Figure 4.11: Effectiveness of QDDH for individual SWA queries under varying result relative-error thresholds ϵ_{thr}, using both the analytical-model-based and the PD-controller-based buffer-size adaptation methods. The input dataset is `AggrDataset1`.

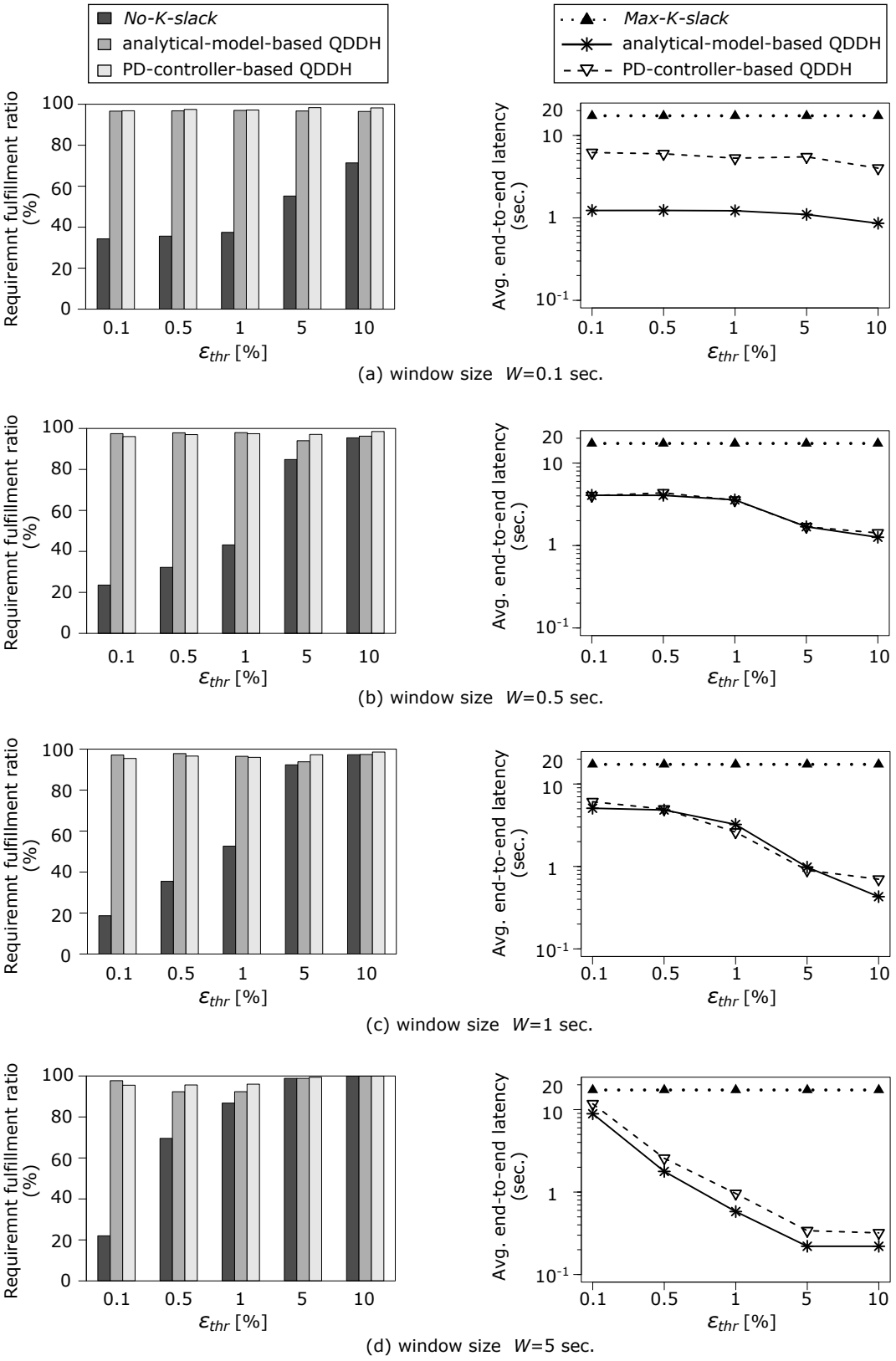


Figure 4.12: Effectiveness of QDDH for individual SWA queries under varying result relative-error thresholds ϵ_{thr}, using both the analytical-model-based and the PD-controller-based buffer-size adaptation methods. The input dataset is `AggrDataset2`.

applied before the next adaptation, whereas *PD-adt* does not. Compared to the *Max-K-slack* approach, *AM-adt* reduced the average end-to-end-latency by at least 48%, and up to 99.98%. For instance, for the query with a window of $W = 10$ seconds, when the result relative-error threshold ϵ_{thr} was 10%, the average end-to-end latency produced by *Max-K-slack* was 17.3 seconds, whereas the average end-to-end latency produced by *AM-adt* was only 0.22 second, which is a significant reduction. *PD-adt* also achieved a latency reduction of at least 32%.

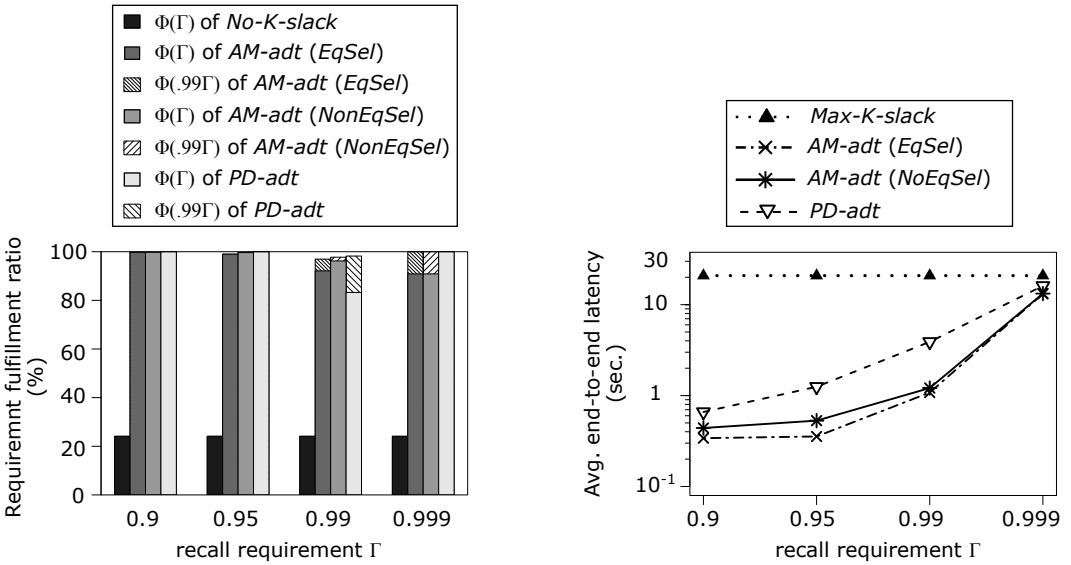
For both *AM-adt* and *PD-adt*, the worst-case latency-reduction occurred when the strictest ϵ_{thr} value was used, i.e., $\epsilon_{thr} = 0.1\%$. More significant latency-reductions were obtained under more moderate ϵ_{thr} values. This phenomenon in turn confirms the potential and the benefit of QDDH.

Effectiveness Results of QDDH for MSWJ

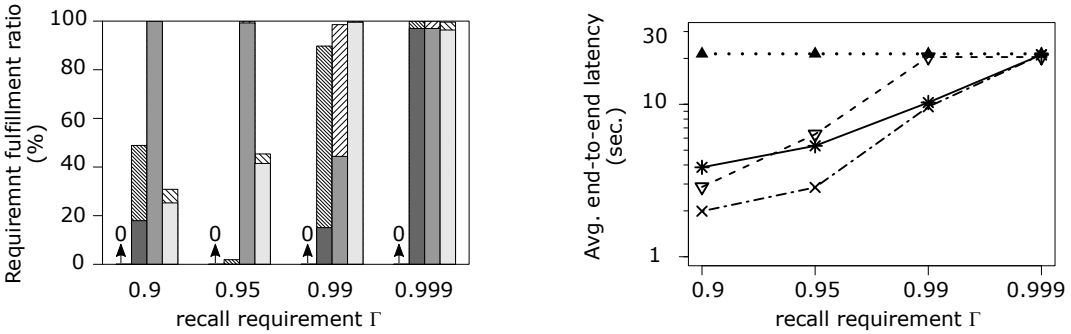
Varying Recall Requirements Figure 4.13 shows the effectiveness results of the proposed instantiation of the QDDH framework for individual MSWJ queries under varying user-specified recall requirements Γ, using both the *AM-adt* (cf. Section 4.2.4) and the *PD-adt* (cf. Section 4.2.5) methods. Recall that *AM-adt* in the instantiation of the QDDH framework for MSWJ queries is based on an analytical model of the recall of the produced join results, $\gamma(L_{adt}, K)$, which uses statistics collected from the past data to make predictions about the future data. Due to the dynamic nature of data streams, it is impossible to make precise predictions; as a result, the derived K-slack buffer sizes may not guarantee a requirement fulfillment ratio $\Phi(\Gamma)$ of 100%. However, a produced recall $\gamma(P_{meas})$ that violates the user-specified recall requirement Γ can be indeed very close to Γ, and is acceptable in most scenarios. Hence, in this experiment, in addition to $\Phi(\Gamma)$, $\Phi(.99\Gamma)$—the percentage of $\gamma(P_{meas})$ measurements that are not lower than Γ by 1%—was measured as well. This experiment also compared the two modeling strategies—*EqSel* and *NonEqSel*—that can be applied in *AM-adt* (cf. Section 4.2.4).

From Figure 4.13, it can be seen that, for both the *EqSel* and the *NonEqSel* modeling strategies, the average end-to-end latency goes up as the recall requirement Γ increases, which again reveals the tradeoff between the latency and the query-result quality. *NonEqSel* produced a bit higher average latency than *EqSel*. The requirement fulfillment ratios $\Phi(\Gamma)$ and $\Phi(.99\Gamma)$ produced by *NonEqSel* were not much higher than those produced by *EqSel* for ($\texttt{JoinDataset}^{\times 2}_{real}$, $Q^{\times 2}$) and ($\texttt{JoinDataset}^{\times 4}_{syn}$, $Q^{\times 4}$), but were significantly higher for ($\texttt{JoinDataset}^{\times 3}_{syn}$, $Q^{\times 3}$). For each ($\texttt{JoinDataset}^{\times i}$, $Q^{\times i}$), $i \in \{2, 3, 4\}$, *NonEqSel* achieved a $\Phi(.99\Gamma)$ of at least 97% for all the examined values of Γ. This result shows that *NonEqSel* is more robust than *EqSel* towards different datasets and join queries. Hence, the rest experiments on the instantiation of the QDDH framework for MSWJ queries used only the *NonEqSel* modeling strategy for the *AM-adt* method.

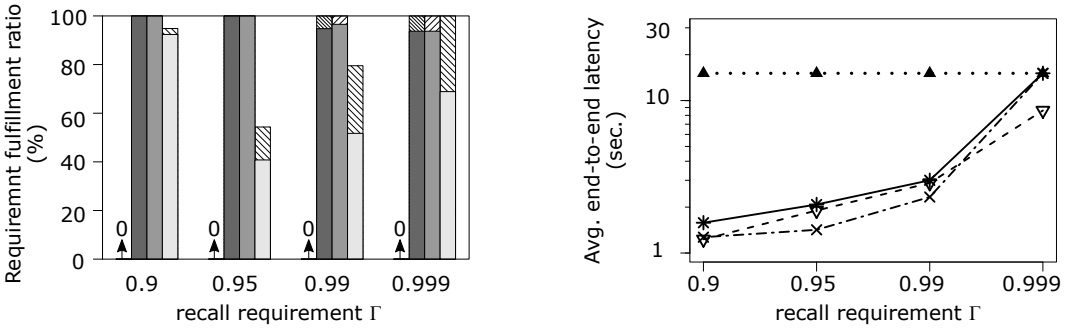
In contrast to in QDDH for SWA, in QDDH for MSWJ, the performance of *PD-adt* is noticeably worse than the performance of *AM-adt* in terms of the achieved requirement fulfillment ratio. For instance, for $\Gamma = 0.95$, $\Phi(.99\Gamma)$ of *PD-adt* were only 45.4% and 54.3% for ($\texttt{JoinDataset}^{\times 3}_{syn}$, $Q^{\times 3}$) and ($\texttt{JoinDataset}^{\times 4}_{syn}$, $Q^{\times 4}$), respectively. The main reason is, as discussed in Section 4.2.5, the performance of the PD controller relies heavily on the accuracy of the measurements of the process variable, which



(a) (**JoinDataset**$^{x2}_{real}$, Q^{x2})



(b) (**JoinDataset**$^{x3}_{syn}$, Q^{x3})



(c) (**JoinDataset**$^{x4}_{syn}$, Q^{x4})

Figure 4.13: Effectiveness of QDDH for individual MSWJ queries under varying recall requirements Γ, using both the analytical-model-based and the PD-controller-based buffer-size adaptation methods.

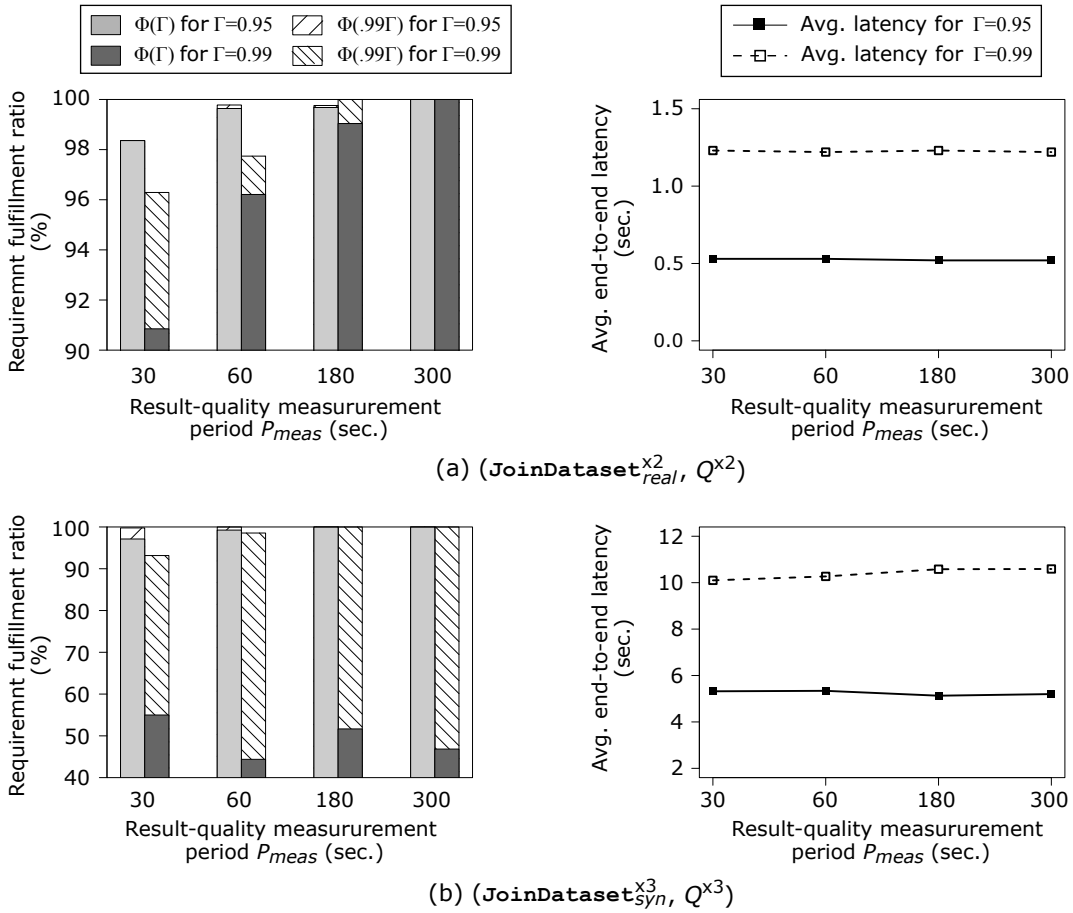


Figure 4.14: Effectiveness of QDDH for individual MSWJ queries under varying result-quality measurement periods P_{meas}, using the analytical-model-based buffer-size adaptation method with the *NonEqSel* modeling strategy.

is the recall $\gamma(P_{meas})$ of join results in QDDH for MSWJ and the window coverage *Cvrg* in QDDH for SWA. It is more difficult to obtain accurate $\gamma(P_{meas})$ measurements than to obtain accurate *Cvrg* measurements; because *Cvrg* is determined by only the numbers of tuples in a single instantaneous window, whereas $\gamma(P_{meas})$ is determined by values of tuples from m ($m \geq 2$) input streams as well as the join condition of the MSWJ query. Although for ($\texttt{JoinDataset}^{\times 2}_{real}$, $Q^{\times 2}$), *PD-adt* and *AM-adt* achieved comparable requirement fulfillment ratios, *PD-adt* produced a higher average end-to-end latency for each examined recall requirement. This result shows that *AM-adt* is more robust than *PD-adt* when dealing with MSWJ queries. Hence, in the rest experiments on the instantiation of theQDDH framework for MSWJ queries, only the *AM-adt* method was used.

Compared to the *Max-K-slack* baseline approach, *AM-adt* with the *NonEqSel* modeling strategy can significantly reduce the average *K*-slack buffer sizes, thus the end-to-end latency, while still honoring the user-specified recall requirement. For instance, even for a high recall requirement $\Gamma = 0.99$, the average latency was reduced by more than 94% (from 20.8 seconds to 1.08 seconds) for ($\texttt{JoinDataset}^{\times 2}_{real}$, $Q^{\times 2}$). For an even higher recall requirement $\Gamma = 0.999$, *AM-adt* still achieved a 36.5% reduction

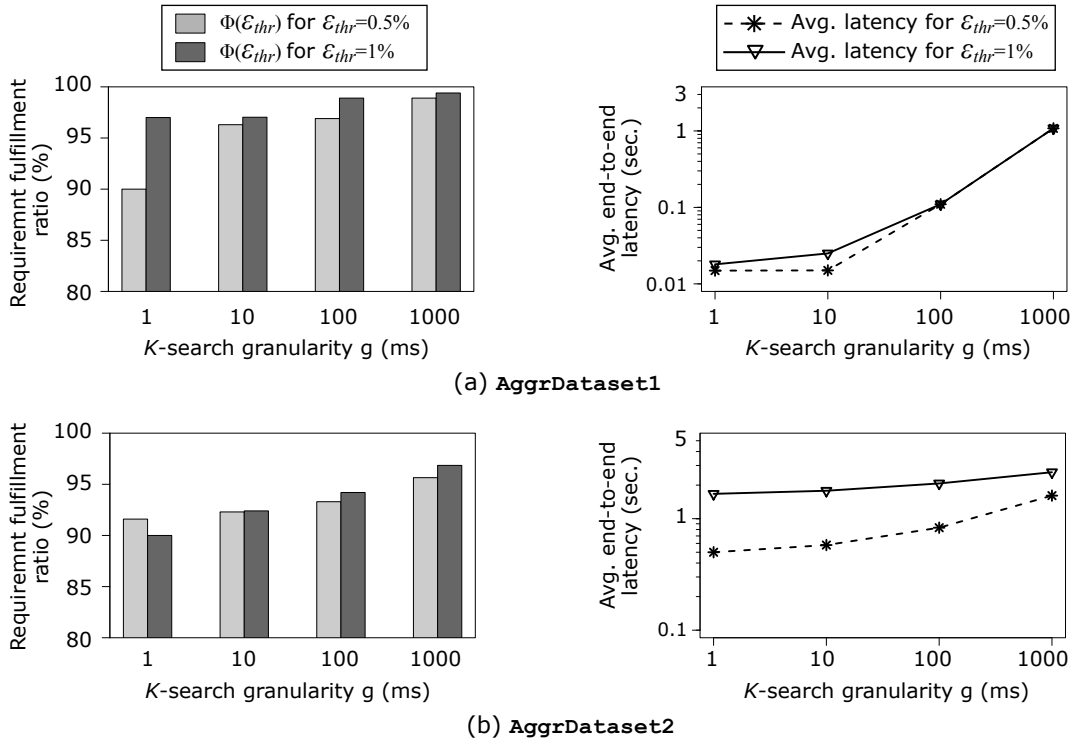


Figure 4.15: Effect of the *K*-search granularity g on the performance of the analytical-model-based buffer-size adaptation method in QDDH for individual SWA queries. A sliding-window SUM query with a window size of $W = 5$ seconds and a window slide of $\beta = 0.1$ second was used.

in the average latency for ($\texttt{JoinDataset}^{\times 2}_{real}$, $Q^{\times 2}$), and degenerated to the *Max-K-slack* approach for ($\texttt{JoinDataset}^{\times 3}_{syn}$, $Q^{\times 3}$) and ($\texttt{JoinDataset}^{\times 4}_{syn}$, $Q^{\times 4}$).

Varying Result-Quality Measurement Periods The next experiment for examining the effectiveness of QDDH for MSWJ focused on studying the effectiveness under varying user-specified result-quality measurement periods P_{meas}. Figure 4.14 shows the results for ($\texttt{JoinDataset}^{\times 2}_{real}$, $Q^{\times 2}$) and ($\texttt{JoinDataset}^{\times 3}_{syn}$, $Q^{\times 3}$) under $\Gamma = 0.95$ and $\Gamma = 0.99$. The other parameters took default values (cf. Table 4.2).

As expected, it is more difficult to obtain high requirement fulfillment ratios $\Phi(\Gamma)$ and $\Phi(.99\Gamma)$ for small values of P_{meas} than for big values of P_{meas}; because the smaller the value of P_{meas}, the slimmer the chance that a low recall of the join results produced within one adaptation interval gets compensated by the recalls produced in the other adaptation intervals that are within the same result-quality measurement period. Nevertheless, the *AM-adt* method with the *NonEqSel* modeling strategy still achieved a $\Phi(.99\Gamma)$ of more than 90% for all examined values of P_{meas}. Similar results were observed for ($\texttt{JoinDataset}^{\times 4}_{syn}$, $Q^{\times 4}$) and other values of Γ.

4.3.4 Effect of Important System Parameters

This section studies the effect of three important parameters in the two QDDH-framework instantiations, including the *K*-search granularity g that is applied in the

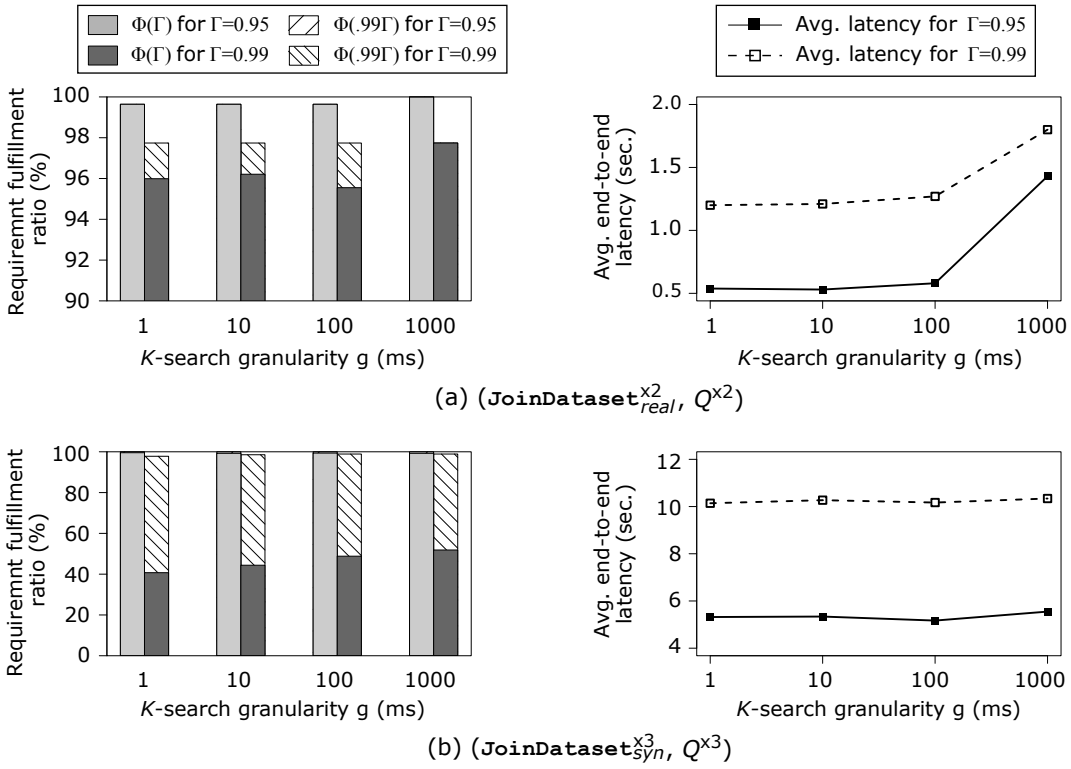


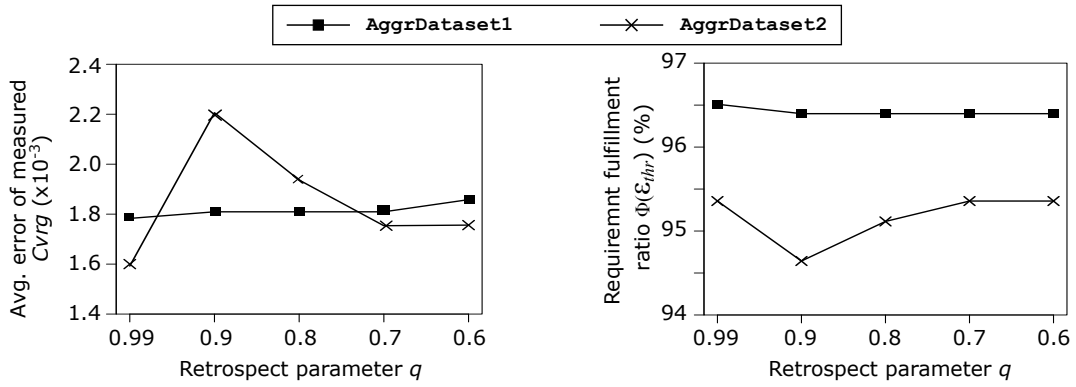
Figure 4.16: Effect of the *K*-search granularity g on the performance of the analytical-model-based buffer-size adaptation method (with *NonEqSel*) in QDDH for individual MSWJ queries.

AM-adt method in both instantiations, the retrospect parameter q that is specific to QDDH for SWA, and the adaptation interval L that is specific to QDDH for MSWJ.

Effect of the *K*-Search granularity g

Recall that in each iteration of the *K*-slack buffer-size adaptation, *AM-adt* searches for the optimal setting of the buffer size by examining possible *K* values incrementally, starting from the value 0 (cf. Algorithm 4.4 and Algorithm 4.7). The increment granularity is g. Figure 4.15 and Figure 4.16 study the effect of the setting of g on the performance of *AM-adt*. The value of g was varied from 1 ms to 1000 ms. The query used in Figure 4.15 is a sling-window SUM query with a window size of $W = 5$ seconds and a window slide of $\beta = 0.1$ second. The other parameters took default values (cf. Table 4.2).

From Figure 4.15 and Figure 4.16, one can observe that as the *K*-search granularity g increases, the average end-to-end latency increases noticeably in all test cases except for (JoinDataset$^{\times 3}_{syn}$, $Q^{\times 3}$). The *K*-slack buffer size required to satisfy the user-specified result-quality requirement for (JoinDataset$^{\times 3}_{syn}$, $Q^{\times 3}$) is larger than that required in the test cases of the other three datasets. This result implies that the value of g has stronger effect in scenarios where satisfying the user-specified result-quality requirement requires a small buffer size than in scenarios where satisfying the result-quality requirement requires a big buffer size.



(a) Error of window-coverage measurements

(b) Produced requirement fulfillment ratio

Figure 4.17: Effect of the retrospect parameter q on the performance of the PD-controller-based buffer-size adaptation method in QDDH for individual SWA queries. A sliding-window SUM query with a window size of $W = 1$ second and a window slide of $\beta = 0.1$ second was used.

Based on the experimental results, $g = 10$ *ms* was chosen empirically as the default setting in the proposed instantiations of the QDDH framework.

Effect of the Retrospect Parameter q in QDDH for SWA

Recall from Section 4.1.4 that the retrospect parameter q is used by the *Window-Coverage Runtime Calculator* in the instantiations of the QDDH framework for SWA queries to measure the coverages of constructed instantaneous windows. The accuracy of these measurements is more critical for the *PD-adt* method than for the *AM-adt* method; because *PD-adt* takes these measurements as values of the process variable, which influence the adaptation decisions produced by the PD controller directly. In contrast, *AM-adt* uses these measurements only to optionally calibrate the window coverages estimated by the analytical model. Hence, this experiment focused on studying the effect of q on the performance of *PD-adt* in QDDH for SWA.

In this experiment, a sling-window SUM query with a window size of $W = 1$ second and a window slide of $\beta = 0.1$ second was used and the result relative-error threshold was $\epsilon_{thr} = 0.1\%$. The value of the retrospect parameter q was varied from 0.99 to 0.6. During each test run, all measured window coverages *Cvrg* were recorded. The map $\mathcal{M}^w$, which maintains the number of missing tuples in each constructed instantaneous window (cf. Section 4.1.6), was not purged, so that true window coverages (denoted by $Cvrg_{true}$) can be computed at the end of each test run. The error $Cvrg_{err}$ of a window-coverage measurement *Cvrg* is computed as the difference between *Cvrg* and its corresponding true window coverage $Cvrg_{true}$, i.e., $Cvrg_{err} = Cvrg - Cvrg_{true}$.

Figure 4.17 shows the resulting average $Cvrg_{err}$ over the whole test run as well as the produced requirement fulfillment ratio $\Phi(\epsilon_{thr})$ under the examined q values. The experimental results for `AggrDataset2` suggest that a smaller value of q does not always lead to a higher average $Cvrg_{err}$ or a lower $\Phi(\epsilon_{thr})$. This non-monotonic behavior is caused by the complex interaction between the measured window coverages and the applied K-slack buffer size in the *PD-adt* method. When a small value of q is

applied, the measured coverage of an instantaneous window is often smaller than its true value, because the *Window-Coverage Runtime Calculator* may have not observed all of the out-of-order tuples that are missing from the instantaneous window (cf. Section 4.1.4). With *PD-adt*, this would lead to a more aggressive decrease of the parameter α, thus the applied K-slack buffer size, compared to the case where a big value of q is applied. However, such an aggressively-decreased buffer size under a small value of q may in turn lead to an earlier buffer-size increase; because more out-of-order tuples would not be sorted correctly by the small buffer, resulting in instantaneous windows with lower coverages. Depending on the disorder characteristics of the input stream, an early buffer increase may be beneficial for dealing with the disorder among upcoming tuples, thereby improving the coverages of the following few instantaneous windows.

Nevertheless, for both `AggrDataset1` and `AggrDataset2`, a higher average window-coverage error $Cvrg_{err}$ leads to a lower requirement fulfillment ratio $\Phi(\epsilon_{thr})$. The result accuracy under $q = 0.99$ is the best for both datasets; hence, $q = 0.99$ was chosen as the default setting in the proposed instantiation of the QDDH framework for SWA queries.

Effect of the Adaptation Interval L in QDDH for MSWJ

Figure 4.18 studies the effect of the adaptation interval L on the performance of the *AM-adt* method in the instantiation of the QDDH framework for MSWJ queries, where L was varied from 0.1 second to 10 seconds. The figure reports results for ($\texttt{JoinDataset}^{\times 2}_{real}$, $Q^{\times 2}$) and ($\texttt{JoinDataset}^{\times 3}_{syn}$, $Q^{\times 3}$) under $\Gamma = 0.95$ and $\Gamma = 0.99$. The other parameters took default values (cf. Table 4.2).

It can be observed that the average end-to-end latency grows noticeably as L increases. This can be explained by the selectivity estimation done in Eq. (4.8). Recall from Section 4.2.4 that the productivity of an out-of-order tuple arrived at the join operator is estimated conservatively as the maximum tuple productivity observed within the last adaptation interval. The maximum tuple productivity observed within a long adaptation interval is often higher than that observed in a short adaptation interval; hence, the estimated selectivity is smaller, which often leads to a smaller outcome of Eq. (4.8). Moreover, with a long adaptation interval, a large value of the buffer size K determined in one iteration of the adaptation is also applied for a longer period of time than with a short adaptation interval.

As L increases, the resulting increase of the applied buffer size K leads to a great increase of $\Phi(\Gamma)$ for ($\texttt{JoinDataset}^{\times 3}_{syn}$, $Q^{\times 3}$) under $\Gamma = 0.99$, but has little effect on the achieved requirement fulfillment ratio for the other three examined cases. In general, $L = 1$ second produced a good tradeoff between the average end-to-end latency and the achieved query-result quality compared with the other four values of L.

4.3.5 Overhead of Buffer-Size Adaptation

The last part of the evaluation studied the overhead of the quality-driven buffer-size adaptation with respect to the runtime, i.e., the time needed to determine the new K-slack buffer size in an individual iteration of the adaptation.

The adaptation time of the *PD-adt* method is nearly constant, at around 25 μs, in all scenarios, regardless of the query type or the result-quality specification. In

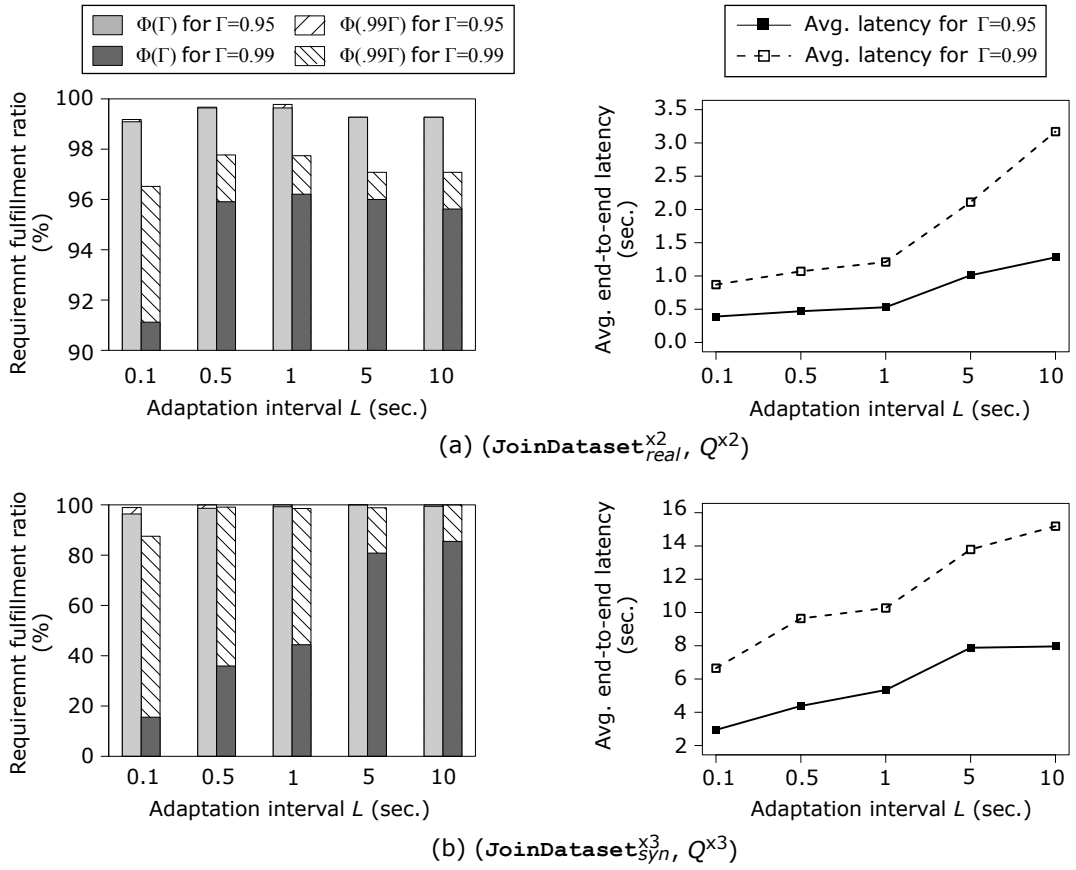
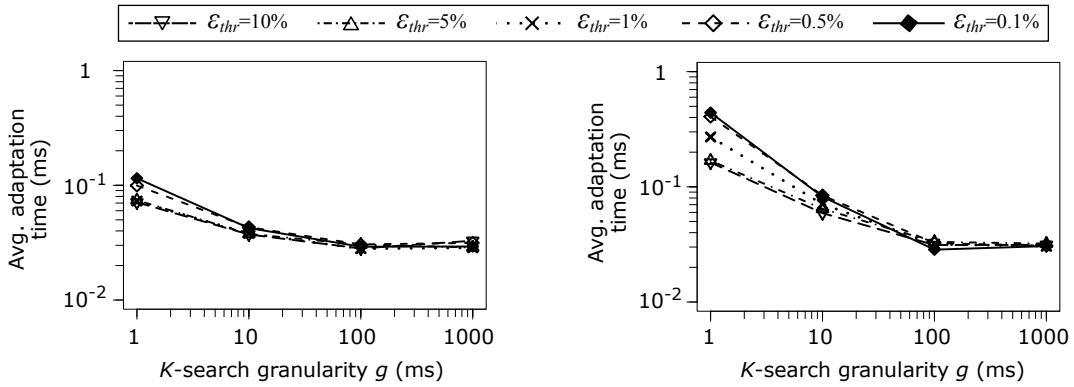


Figure 4.18: Effect of the adaptation interval L on the performance of the analytical-model-based buffer-size adaptation method (with *NonEqSel*) in QDDH for individual MSWJ queries.

contrast, the adaptation time of the *AM-adt* method is influenced by the number of input streams m, the user-specified result-quality requirement, i.e., ϵ_{thr} or Γ, and the K-search granularity g. The more number of input streams that a query involves, the higher complexity of the corresponding analytical model; and the smaller the K-search granularity, the more number of iterations *AM-adt* needs to run to reach a specific optimal QDDH buffer size κ (cf. Algorithm 4.4 and Algorithm 4.7).

Figure 4.19 shows the experimental results of the instantiation of the QDDH framework for SWA queries, where a sliding-window SUM query with a window size of $W = 1$ second and a window slide of $\beta = 0.1$ second was executed, under different combinations of ϵ_{thr} and g values. The other parameters took default values (cf. Table 4.2). The query was executed three times over each `AggrDataset`. For each combination of ϵ_{thr} and g, the average time consumed by an individual adaptation of the K-slack buffer size over all three runs of the query under that combination was recorded and reported in the figure. Figure 4.20 shows the experimental results of the QDDH-framework instantiation for MSWJ queries. Analogously, different combinations of Γ and g values were evaluated, over each `JoinDataset`.

As expected, the adaptation time goes down as the K-search granularity g increases and as the result-quality requirement gets lower (i.e., as the result relative-er-



(a) **AggrDataset1** (b) **AggrDataset2**

Figure 4.19: Time needed by the analytical-model-based buffer-size adaptation method to derive a new K-slack buffer size in an individual adaptation iteration in QDDH for individual SWA queries. A sliding-window SUM query with a window size of $W = 1$ second and a window slide of $\beta = 0.1$ second was used.

ror threshold ϵ_{thr} increases and as the recall requirement Γ decreases). In Figure 4.19, the average adaptation time was below 1 ms for all examined combinations of ϵ_{thr} and g for both datasets. In Figure 4.20, when $g \geq 10$ ms, the average adaptation time was below 5 ms for the highest examined value of Γ on all datasets. The average adaptation time for lower Γ values was even smaller—below 1 ms. In the implementation of the instantiations of the QDDH framework, the *Buffer Manager* and window-based query operators run in separate threads. Hence, the buffer-size adaptation time indeed overlaps with the processing time of the aggregate or the join.

4.3.6 Summary of Experimental Results

In summary, the baseline results presented in Section 4.3.2 show that there is an inevitable tradeoff between the end-to-end latency and the query-result quality in disorder handling; and naive disorder handling approaches that make extreme tradeoffs between these two requirements can lead to either a high latency, or a considerably-low query-result quality. These baseline results also confirm that the QDDH concept proposed in this dissertation has a practical application area.

Section 4.3.3 evaluated each of the two QDDH-framework instantiations proposed in this chapter with different datasets, queries, and settings of respective result-quality requirements. The experimental results show that the two instantiations with the proposed analytical-model-based buffer-size adaptation method are effective in terms of quality-driven latency minimization. Compared to the *Max-K-slack* baseline approach, QDDH reduced the applied K-slack buffer sizes, thus the end-to-end latency, significantly (up to 99.98% across all examined cases), while respecting user-specified result-quality requirements. It was also shown that the analytical-model-based adaptation method is more robust than the PD-controller-based adaptation method, especially for MSWJ queries.

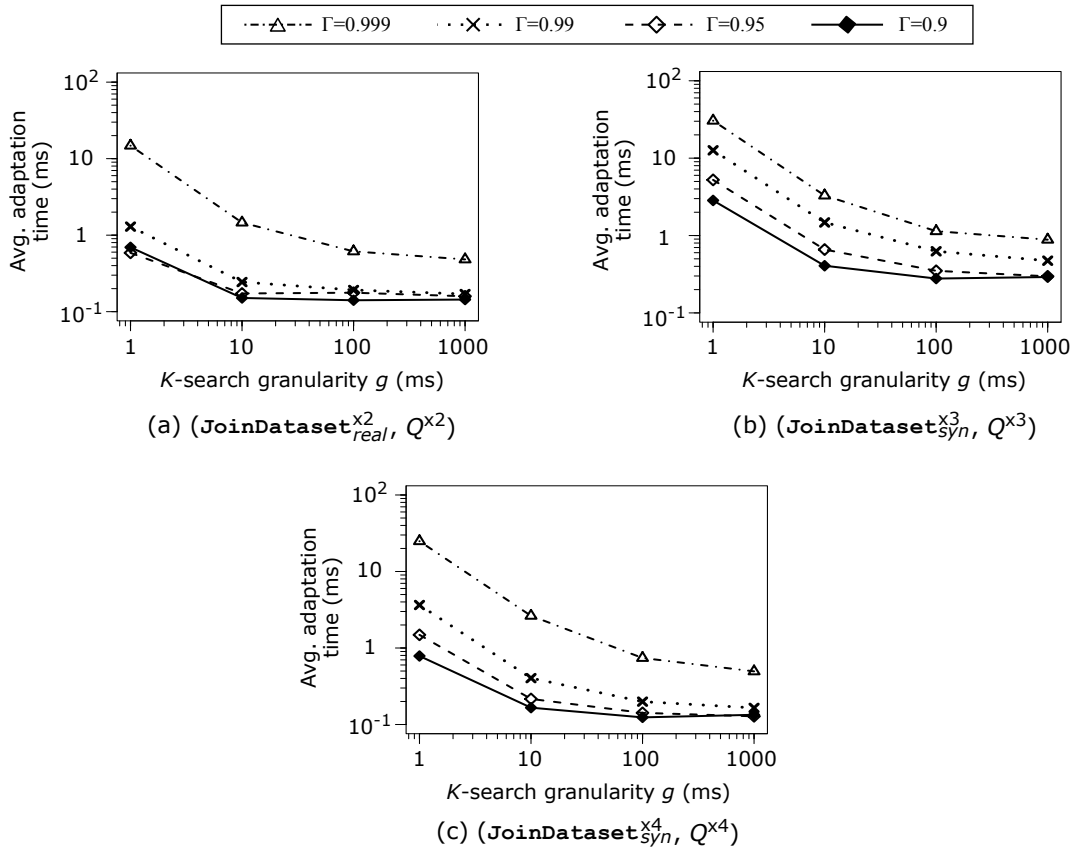


Figure 4.20: Time needed by the analytical-model-based buffer-size adaptation method to derive a new *K*-slack buffer size in an individual adaptation iteration in QDDH for individual MSWJ queries.

The overhead results presented in Section 4.3.5 show that the objective of QDDH can be achieved with a low runtime overhead, which confirms the practicability of the proposed QDDH approach.

4.4 Summary

This chapter described instantiations of the generic QDDH framework introduced in Chapter 3 for individual SWA and MSWJ queries. For each instantiation, it was discussed in detail how to specialize the two quality-driven buffer-size adaptation methods—the analytical-model-based method and the PD-controller-based method—that were introduced in Section 3.4 based on the respective query type. Experimental results showed the effectiveness of the two proposed instantiations with respect to quality-driven latency minimization. Compared to the state of the art, both instantiations can significantly reduce the *K*-slack buffer sizes applied for disorder handling, thus the end-to-end latency, while still providing the user-desired query-result quality.

Note that for streams having a large amount of out-of-order tuples with large delays, the end-to-end latency produced under QDDH could still be high under high

result-quality requirements. To reduce the end-to-end latency for queries executed over this type of streams, the proposed QDDH approach can be combined with the speculation-based disorder handling approach (cf. Section 3.5) in a similar way as is done in the work by Krishnamurthy et al. [Kri+10]. Specifically, an upper bound K_{ub} for the K-slack buffer size can be set in the *Buffer Manager*. The *Buffer Manager* then deals with out-of-order tuples whose delays are not larger than K_{ub} using the proposed QDDH approach, and deals with out-of-order tuples whose delays are larger than K_{ub} using speculation and retraction.

5

Quality-Driven Disorder Handling for Concurrent Queries with Shared Operators

Chapter 4 discussed instantiations of the generic QDDH framework for individual SWA and MSWJ queries. However, a natural workload pattern in a DSPS is to execute multiple queries concurrently over a collection of input streams. Moreover, when executing multiple queries concurrently, a commonly-used technique for improving the system performance is to share computations among the concurrent queries [Che+00; Hir+14].

Based on the work presented in Chapter 4, this chapter takes one step further to consider QDDH for concurrent queries with shared operators. As in Chapter 4, this chapter focuses on SWA and MSWJ queries, which are the predominant query types used in stream-based applications. In this chapter, in addition to the window-based aggregate or join operation, each query may filter its input streams using arbitrary selection predicates (cf. Section 2.1.2). This chapter focuses on the case of sharing the computation across selection predicates; namely, selection predicates that are common in different queries will be evaluated by shared filters, i.e., shared selection operators. After defining the problem of QDDH in the context of concurrent queries with shared query operators in Section 5.1, Section 5.2 gives an overview of the instantiation of the generic QDDH framework in this context. Subsequently, Section 5.3, Section 5.4, and Section 5.5 drill down to the details of this QDDH-framework instantiation. Section 5.6 presents the evaluation results and Section 5.7 discusses related work that is relevant to the content of this chapter.

5.1 Introduction

Motivation

The logical query plan that represents the query-operator network of concurrent queries obtained after exploiting the sharing opportunities is called a *global logical query plan*, or for short, a *global query plan*, denoted by G^{glob}. The work presented in this chapter focuses on exploiting the sharing opportunities among the selection

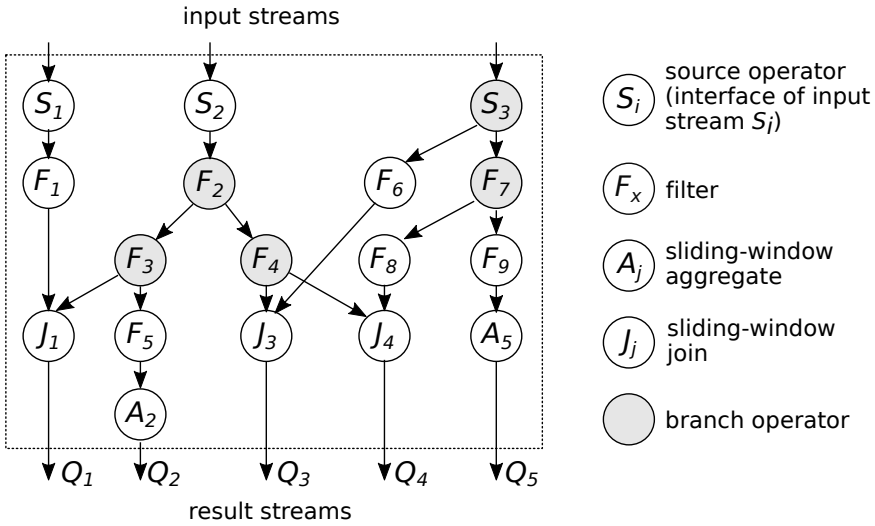


Figure 5.1: A global query plan constructed after fully exploiting the sharing opportunities among the selection predicates of five concurrent queries.

predicates in concurrent queries. Let Q_j denote the j-th query contained in a global query plan. Figure 5.1 shows an example of a global query plan, which consists of two MSWJ queries and three SWA queries. In this global query plan, window operators and the final R2S operators, *Istream* (cf. Section 2.1.2), are omitted for brevity.

Given a global query plan G^{glob} and the user-specified result-quality requirement for each query contained in G^{glob}, the objective of QDDH is to *minimize the end-to-end latency of each query, while subjecting the result quality of each query to its associated user-specified result-quality requirement*. A naive solution to achieve this objective is *unshared QDDH*. Namely, for each query Q_j contained in G^{glob}, the intra-stream disorder handling components—K-slack buffers—are placed right before the window-based join or aggregate operator in Q_j (i.e., right above the *leaf* nodes in G^{glob}); and the sizes of these buffers are dynamically adjusted in a quality-driven way, independent of the size adjustment of the K-slack buffers applied in any other query in G^{glob}. However, such an unshared QDDH may result in duplicate storing and sorting of the same stream tuples, which is a waste of the memory resources of a DSPS. To illustrate this, let us suppose that the selectivities of the filters F_8 and F_9 in Figure 5.1 are both 0.9. Then, with unshared QDDH, at least 80% ($= 2 \times 0.9 - 1$) of the output tuples of the filter F_7 need to be handled by both the K-slack buffer for the query Q_4 and the K-slack buffer for the query Q_5.

One natural solution to avoid the duplicate disorder handling described above, thereby saving the memory resources of a DSPS, is *shared QDDH*; namely, sharing K-slack buffers across multiple queries. For the example discussed above, shared QDDH means placing a K-slack buffer below the filter F_7, rather than below the filters F_8 and F_9 redundantly. As a result, the output of F_7 is sorted only once, and the sorting effort is shared between the query Q_4 and the query Q_5.

An operator v in a global query plan G^{glob} that has more than one child is referred to as a *branch operator*. In general, shared disorder handling could happen right below any branch operator in G^{glob}, and even happen concurrently right below multiple branch operators. However, for any branch operator, sharing the disorder handling

of the operator's output does not always lead to a lower memory consumption. For instance, when the selectivities of the filters F_8 and F_9 in Figure 5.1 are both 0.1, then doing shared disorder handling right below the filter F_7 has a higher memory consumption than doing unshared disorder handling right below each of F_8 and F_9. Moreover, as will be shown later in this chapter, naive sharing of K-slack buffers may unnecessarily increase the end-to-end latency of queries that have low result-quality requirements, whereas smart sharing of K-slack buffers has a higher overhead of runtime adaptation than naive sharing of K-slack buffers when switching to a new configuration of the buffers. Hence, when doing QDDH for concurrent queries that have shared query operators, it is not obvious *how to place K-slack buffers within the global query plan, so that the objective of QDDH is achieved with a minimum memory consumption*. This is especially true when the global query plan contains a large number of branch operators.

Problem Formulation

Given n SWA and MSWJ queries $\mathbb{Q} = \{Q_1, Q_2, \ldots, Q_n\}$, which involve m input streams $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$, a global query plan G^{glob} like the one shown in Figure 5.1 can be constructed. Specifically, selection predicates applied on input streams are pulled above any join or aggregate operator, and the sharing opportunities among all selection predicates are fully exploited. Moreover, on the basis of sharing, selection predicates are fused wherever possible. As a result, window-based join or aggregate operators only appear as leaves in the global query plan G^{glob}; and any non-branch filter in G^{glob} does not again have a filter as a child. Recall from Section 2.1.2 that a source operator acts as an interface of an external input stream. In this chapter, a source operator shares the same notation as the corresponding input stream because of the one-to-one relationship between them.

Given a global query plan constructed in the above way, the objective of *multi-query QDDH* is defined as *minimizing both the end-to-end latency incurred by K-slack buffers for each query, and the overall memory consumption of all K-slack buffers, while respecting the user-specified result-quality requirement for each query*. Recall from Chapter 4 that the minimum K-slack buffer size required to satisfy the user-specified result-quality requirement of an individual query is referred to as the *optimal QDDH buffer size* for that query and is denoted by κ. In this chapter, κ_j is used to denote the optimal QDDH buffer size for the query Q_j.

This chapter does not consider sharing computations among multiple window-based join operators or multiple window-based aggregate operators. In the context of QDDH, sharing computations among window-based operators using existing approaches (e.g., [Gui+11; KWF06; Wan+06]) may violate the objective of quality-driven latency minimization. To explain this, suppose that based on the window and the operator semantics, the window-based operator in a query Q_k can be evaluated by further processing the results of the window-based operator in another query Q_j. If, however, Q_k had a much lower user-specified result-quality requirement than Q_j, thus requiring a smaller optimal QDDH buffer size than Q_j, then sharing the computation of Q_j to further evaluate Q_k would essentially enforce Q_k to apply the same K-slack buffer size as Q_j, which increases the end-to-end latency of Q_k unnecessarily. How to adapt the existing approaches for sharing window-based operators in the context of QDDH requires further investigation.

To handle the disorder within an input stream S_i of any query $Q_j \in \mathbb{Q}$, one or more K-slack buffers can be placed along the path from the source operator S_i to the window-based join or aggregate operator in Q_j. Let B represent an individual K-slack buffer. Each buffer B has two important properties: (1) the *buffer size* $K(B)$, and (2) the *output targets* $\mathcal{O}_{tgt}(B)$, which is defined as the set of operators that consume the output of the buffer B. In the remainder of this chapter, C^{glob} is used to denote a *global K-slack configuration* for a global query plan G^{glob}. A global K-slack configuration C^{glob} is defined as a possible placement of K-slack buffers within the global query plan G^{glob}, along with the property settings (i.e, the buffer size $K(B)$ and the output targets $\mathcal{O}_{tgt}(B)$) of each placed buffer. To achieve the previously-defined objective of multi-query QDDH, a C^{glob} that satisfies the two conditions below needs to be found:

1. *Latency minimization:* For each query Q_j in G^{glob}, the summed size of the K-slack buffers placed along any path from a source operator involved in Q_j to the window-based aggregate or join operator in Q_j does not exceed the optimal QDDH buffer size κ_j of Q_j.

2. *Memory minimization:* The memory cost of C^{glob}, denoted by $mem(C^{glob})$, is not larger than that of any other global K-slack configuration for G^{glob}. The memory cost $mem(C^{glob})$ at any point in time is measured as the number of tuples kept in all K-slack buffers in C^{glob}.

A global K-slack configuration C^{glob} for a global query plan G^{glob} that satisfies both conditions above is called an *optimal global QDDH K-slack configuration for* G^{glob}.

Denote the subplan of a global query plan that is rooted at a source operator S_i by G^i. Because the disorder handling of one input stream has no influence on the disorder handling of the other input streams, the task of finding the optimal C^{glob} for a global query plan G^{glob} can be broken down to sub-tasks of finding the optimal K-slack configuration for each subplan G^i of G^{glob}. A K-slack configuration for the subplan G^i is denoted by C^i, and the memory cost of C^i is denoted by $mem(C^i)$.

Contributions

Specifically, the work presented in this chapter makes the following contributions:

- The notion of *K-slack chain* is introduced for sharing the disorder handling of the output of a branch operator in a global query plan. A K-slack chain can be shared by queries having different result-quality requirements, i.e., different optimal QDDH buffer sizes, without enforcing queries that have low result-quality requirements to use larger-than-necessary K-slack buffer sizes. (Section 5.3)

- Two algorithms—*GREEDY* and *OPT*—are proposed for determining the global K-slack configuration for a global query plan G^{glob}. The algorithm *GREEDY* trades the memory optimality of the determined K-slack configuration for low computational cost, and does not enumerate all possible K-slack configurations for G^{glob}. The algorithm *OPT* can find the memory-optimal K-slack configuration to achieve the objective of QDDH for G^{glob}, yet without doing exhaustive enumeration of all possible K-slack configurations either. (Section 5.4)

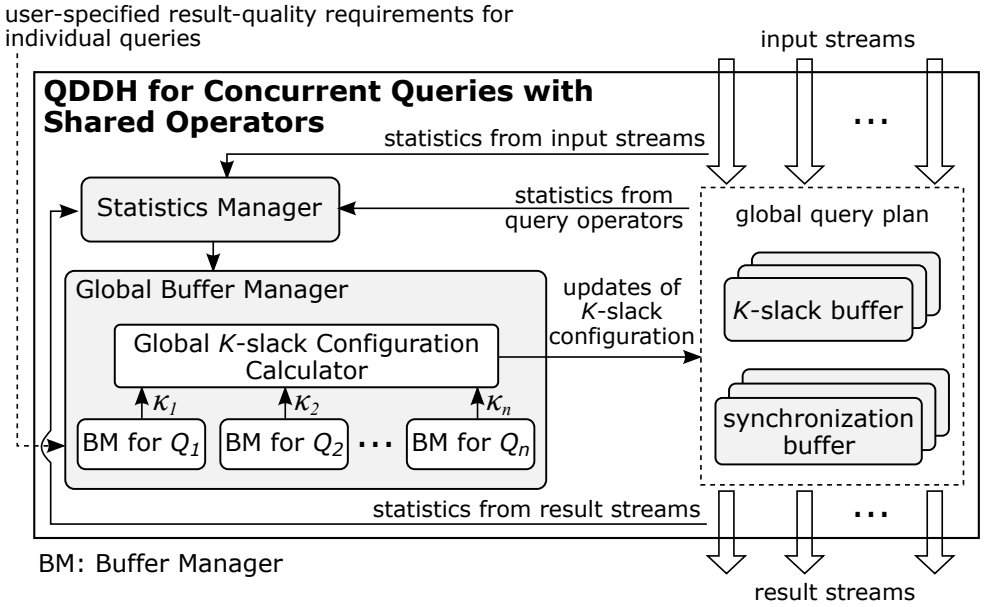


Figure 5.2: Instantiation of the buffer-based QDDH framework for concurrent SWA and MSWJ queries with shared source and filter operators.

- Different strategies for triggering the adaptation of the applied *K*-slack configuration at the query runtime are introduced. Moreover, the overhead of performing semantics-preserving runtime adaptation is analyzed; and scenarios where the overhead of runtime adaptation can be reduced by reusing buffers from the old configuration are discussed. (Section 5.5)

5.2 QDDH-Framework Instantiation Overview

Figure 5.2 gives an overview of the instantiation of the generic QDDH framework for a global query plan of concurrent SWA and MSWJ queries with shared source and filter operators. The *Buffer Manager* in the generic framework (cf. Figure 3.5) becomes a *Global Buffer Manager* in Figure 5.2, which consists of a *Global K-slack Configuration Calculator* and *n Buffer Manager*s for the n queries contained in the global query plan.

The *Buffer Manager* for each query Q_i in a global query plan is responsible for deriving the optimal QDDH buffer size κ_i for the query Q_i dynamically, in the way as described in Chapter 4. In this chapter, it is assumed that each *Buffer Manager* uses the analytical-model-based buffer-size adaptation method to derive the optimal QDDH buffer size for the corresponding query. Recall that for a SWA query, κ is updated whenever a new instantaneous window has been constructed at the SWA operator in the query (cf. Section 4.1.2); and for an MSWJ query, κ is updated every L time units, where the adaptation interval L is a configurable system parameter (cf. Section 4.2.2). The updates of each κ_i ($i \in [1, n]$) are forwarded to the *Global K-slack Configuration Calculator*, which decides whether a runtime adaptation of the *K*-slack configuration applied in the global query plan needs to be triggered, and computes the new configuration if the decision is positive. Note that a global *K*-slack configuration C^{glob} is a combination of the *K*-slack configurations C^i for all subplans

G^i of G^{glob}. The solution for determining the optimal K-slack configuration for a subplan G^i is generic to all subplans; hence, in the following sections, the proposed solution is illustrated only for an individual subplan G^i.

In addition to the statistics required by the *Buffer Manager* for each individual query in the global query plan, the *Statistics Manager* in Figure 5.2 also monitors and maintains selectivities of the filters in the global query plan. These filter selectivities are used by the *Global K-slack Configuration Calculator* to compute new K-slack configurations.

5.3 Shared Disorder Handling Using *K*-Slack Chain

Before describing the behavior of the *Global K-slack Configuration Calculator* in detail, this section introduces the concept of *K-slack chain*. A K-slack chain is applied right below a branch operator in a global query plan G^{glob} to perform shared disorder handling for the output of the branch operator, without violating the objective of latency minimization for each query that shares the branch operator. For ease of illustration, let us consider a simple subplan G^1 as shown in Figure 5.3a. This subplan contains only one branch operator S_1 and does not contain filter operators. Assume that the optimal QDDH buffer sizes of the queries contained in G^1 satisfy $\kappa_3 < \kappa_1 < \kappa_2$. Moreover, let $\mathcal{Q}(v)$ denote the set of queries in a global query plan that share the operator v.

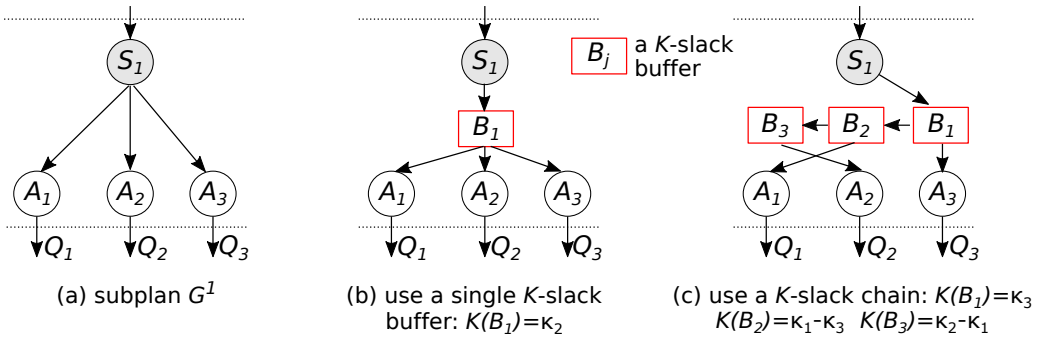


Figure 5.3: Shared disorder handling within a subplan G^i that does not contain filter operators: single K-slack buffer versus K-slack chain. (Assume that $\kappa_3 < \kappa_1 < \kappa_2$.)

For the subplan G^1 in Figure 5.3a, unshared disorder handling of the stream S_1 means placing a separate K-slack buffer above each aggregate operator A_i, $i \in \{1,2,3\}$, in G^1. Recall from Section 5.1 that the memory cost of a K-slack configuration is defined as the number of tuples kept in all K-buffers in the configuration. Based on this definition, the average memory cost of the above K-slack configuration under unshared disorder handling can be determined as $r_1 \cdot (\kappa_1 + \kappa_2 + \kappa_3)$, where r_1 represents the average tuple arrival rate of the stream S_1. Doing shared disorder handling for the stream S_1 can lead to lower memory cost. A naive solution of shared disorder handling is to place a single K-slack buffer below the source operator S_1 (Figure 5.3b). To meet the user-specified result-quality requirement of each query in G^1, the size of this buffer, i.e., the size of B_1 in Figure 5.3b, must not be smaller than the greatest value among κ_1, κ_2, and κ_3, which is κ_2 in this example. However, setting the size of B_1 to κ_2 means that a larger-than-necessary buffer is applied for

the queries Q_1 and Q_3, which violates the *latency-minimization* condition defined in Section 5.1.

To overcome the above drawback of a single K-slack buffer, this dissertation proposes to use a chain of K-slack buffers as shown in Figure 5.3c. This proposal is based on the following property of the K-slack algorithm: *the disorder-handling effect of a single K-slack buffer of size k is equivalent to the disorder-handling effect of a chain of K-slack buffers whose summed size is k*. In Figure 5.3c, the size of the first buffer B_1 in the K-slack chain is set based on the smallest optimal QDDH buffer size among the queries in $\mathcal{Q}(S_i)$, i.e., $K(B_1) = \kappa_3$. The output tuples of B_1 are forwarded to both the operator A_3 and the second K-slack buffer B_2, so that they can be processed by A_3 without being delayed further, and at the same time be handled further by B_2 to meet the result-quality requirements of the queries Q_1 and Q_2. The size of B_2 is $K(B_2) = \kappa_1 - \kappa_3$, i.e., the difference between the second smallest κ value and the smallest κ value among $\{\kappa_i | i = 1, 2, 3\}$. The output tuples of B_2 are forwarded to both the operator A_1 and the last K-slack buffer B_3. The size of B_3 is $K(B_3) = \kappa_2 - \kappa_1$, i.e., the difference between the greatest κ value and the second greatest κ value among $\{\kappa_i | i = 1, 2, 3\}$. The output tuples of B_3 are forwarded to the operator A_2 only.

The total size of the buffers in the K-slack chain in Figure 5.3c is $(\kappa_2 - \kappa_1) + (\kappa_1 - \kappa_3) + \kappa_3 = \kappa_2$. Hence, it has the same, which is also the optimal, memory cost as the single K-slack buffer in Figure 5.3b. However, the K-slack chain overcomes the drawback of the single K-slack buffer and satisfies the condition of latency minimization.

In general, for a branch operator v that has no filter child, a K-slack chain of length u can be used right below v to perform condition-satisfying shared disorder handling, where u is the number of distinct κ values among the queries in $\mathcal{Q}(v)$. The K-slack chain can be built as follows: Sort the list of distinct κ values in increasing order. Let κ^l represent the l-th κ value in the sorted list. For the l-th ($l \in [1, u]$) buffer B_l in the K-slack chain, the buffer size $K(B_l)$ is set to κ^1 if $l = 1$, and to $\kappa^l - \kappa^{l-1}$ otherwise. The output target $\mathcal{O}_{tgt}(B_l)$ of the buffer B_l includes each child of the branch operator v, whose containing query Q_j has an optimal QDDH buffer size κ_j that satisfies $\kappa_j == \kappa^l$.

5.4 Memory-Optimal QDDH

This section presents the proposed solution for finding the optimal K-slack configuration for a subplan G^i in the general case (i.e., G^i could have multiple branch operators which may be source or filter operators), based on the optimal QDDH buffer sizes derived for each individual query contained in G^i. First, Section 5.4.1 describes how to find the optimal *local K-slack configuration*, $C^{(v)}$, for an individual branch operator v. Section 5.4.2 then presents the solution for an entire subplan G^i.

5.4.1 Solution for Individual Branch Operators

In Section 5.3, it was shown that for a branch operator v that does not have any filter child, shared disorder handling using a K-slack chain has lower memory cost than unshared disorder handling. However, this is not always true when the branch operator v has filter children. Depending on the selectivities of the filter children, it

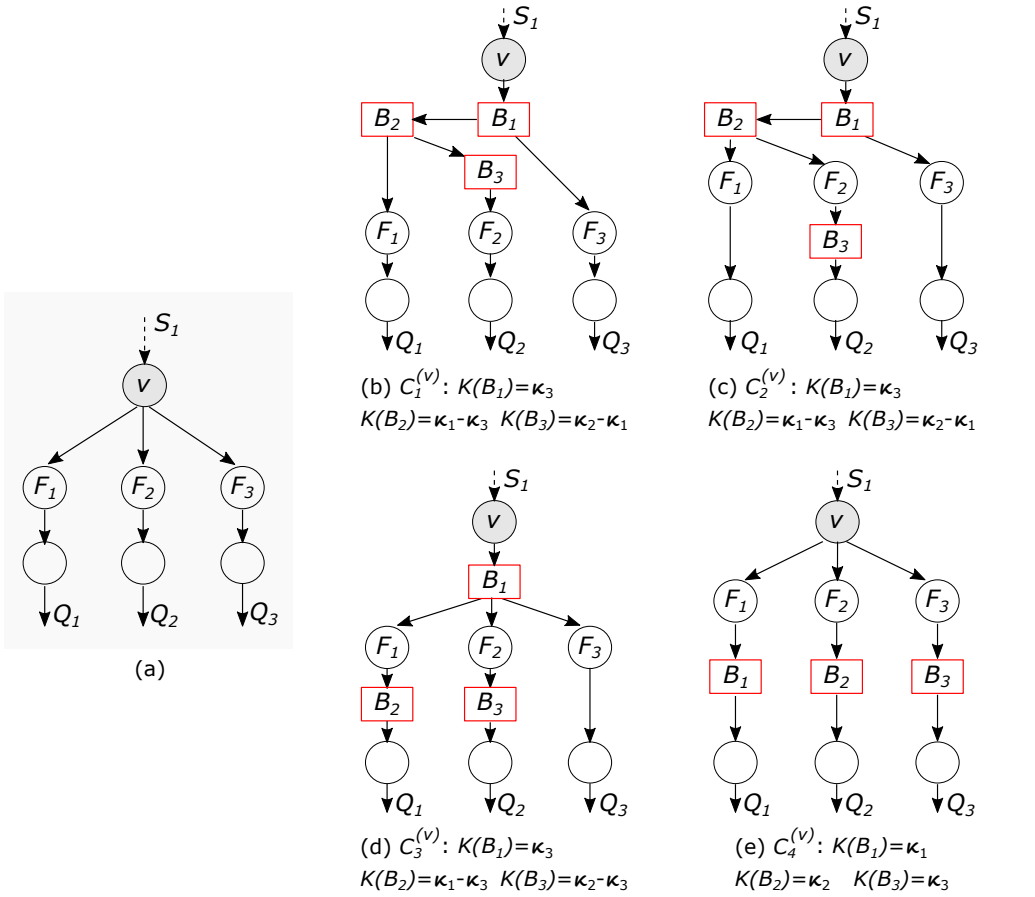


Figure 5.4: All possible memory-optimal local K-slack configurations for a branch operator. Assume that the optimal QDDH buffer sizes of the queries satisfy $0 < \kappa_3 < \kappa_1 < \kappa_2$.

may be more memory-efficient to push buffers at the tail of the K-slack chain down below certain filter children of the branch operator v.

To illustrate this, let us consider the branch operator v in Figure 5.4a, which has three filter children. For the moment, let us assume that each child of v, denoted by v_c, is not again a branch operator. The scenarios where this assumption is relaxed will be discussed in Section 5.4.2. Without loss of generality, assume that the relationship between the optimal QDDH buffer sizes of the three queries in Figure 5.4 is $0 < \kappa_3 < \kappa_1 < \kappa_2$. Figure 5.4b–5.4e show all possible optimal local K-slack configurations for the branch operator v under this assumed relation between κ_1, κ_2, and κ_3. From the configuration $C_1^{(v)}$ to the configuration $C_4^{(v)}$, the buffers in the K-slack chain in $C_1^{(v)}$ are pushed down below the filter children of the branch operator v one by one, starting from the tail buffer. The configuration $C_4^{(v)}$ is indeed doing unshared disorder handling. The specific size of a K-slack buffer in each $C_i^{(v)}$ ($i \in \{1,2,3,4\}$) depends on the position of the buffer within the subplan. Configurations $C_1^{(v)}$ to $C_4^{(v)}$ all meet the condition of latency minimization defined in Section 5.1, but have different memory costs. Each of $C_i^{(v)}$ ($i \in \{1,2,3,4\}$) is referred to as a *candidate*

local K-slack configuration for the operator v. Which one of them is the actual optimal configuration, i.e., the condition of memory minimization is met as well, depends on the selectivities of the filters F_1, F_2, and F_3 in Figure 5.4a.

In general, the total number of candidate local K-slack configurations for a branch operator v is between 1 and $N_F(v)+1$, where $N_F(v)$ is the number of filter children that the branch operator v has. The lower bound, 1, is reached when the branch operator v has a non-filter child, whose containing query has the greatest optimal QDDH buffer size κ among all queries in $\mathcal{Q}(v)$. The reason is that when such a child v_c of v exists, to meet the condition of latency-minimization, the child v_c must be included in the output targets of the tail buffer in the K-slack chain that is put right below v. Because v_c is not a filter, the tail buffer in the K-slack chain cannot be pushed down below v_c. As a result, although it is possible to push down the other buffers in the K-slack chain whose output targets are filter children of v, the summed size of the K-slack chain placed right below v remains the same. Hence, the push-down of the other buffers in the K-slack chain cannot lead to a lower memory cost.

The upper bound of the number of candidate local K-slack configurations for a branch operator v, $N_F(v)+1$, is reached when the containing query of every filter child of v has such an optimal QDDH buffer size that it is greater than the optimal QDDH buffer sizes of the containing queries of all non-filter children of v. The reason is that, in this case, when no buffer in the K-slack chain placed right below v is pushed down below any filter child of v, then a buffer in the chain, whose output targets include a filter child of v, must be after any other buffer in the chain, whose output targets include a non-filter child of v. Namely, the K-slack chain can be viewed as a concatenation of two parts, where the output targets of each buffer in the first part include at least one non-filter child of v, and the output targets of each buffer in the second part include only filter children of v. Buffers in the second part can be pushed down below their respective output targets one by one, in the way as shown in Figure 5.4b–5.4e. Since the branch operator v has $N_F(v)$ filter children, there are at most $N_F(v)$ buffers in the second part of the K-slack chain. Hence, the total number of options for pushing buffers in the K-slack chain down, whose resulting memory costs could be the minimum, is at most $N_F(v)+1$.

Let f_x represent the selectivity of a filter F_x, and $r^{(v)}$ represent the average output rate of the branch operator v in Figure 5.4. The naive approach to find the memory-optimal local K-slack configuration for v is to calculate the memory costs of all four candidate configurations using the equations below, and then compare the calculated memory costs[1]. The selectivities of the filters in the query plan can be monitored at the query runtime.

$$
\begin{aligned}
mem(C_1^{(v)}) &= r^{(v)} \cdot \kappa_2 \\
mem(C_2^{(v)}) &= r^{(v)} \cdot (\kappa_1 + f_2 \cdot (\kappa_2 - \kappa_1)) \\
mem(C_3^{(v)}) &= r^{(v)} \cdot (\kappa_3 + f_1 \cdot (\kappa_1 - \kappa_3) + f_2 \cdot (\kappa_2 - \kappa_3)) \\
mem(C_4^{(v)}) &= r^{(v)} \cdot (f_1 \cdot \kappa_1 + f_2 \cdot \kappa_2 + f_3 \cdot \kappa_3)
\end{aligned}
$$

[1]Note that, actually, the comparison of the memory costs can be done regardless of the value of $r^{(v)}$.

Algorithm 5.1 Determine the optimal local K-slack configuration for a branch operator v that does not have any child that is again a branch operator.

1: $\mathcal{K}(v) \leftarrow$ distinct values in $\{\kappa_j | Q_j \in \mathcal{Q}(v)\}$
2: *localBufSize* $\leftarrow 0$ ▷ the summed size of K-slack buffers placed right below v
3: *sumOfSel* $\leftarrow 0$
4: **while** $\mathcal{K}(v) \neq \emptyset$ **do**
5: $\quad \kappa_{max} \leftarrow max\{\kappa | \kappa \in \mathcal{K}(v)\}$
6: $\quad$ **for** each child v_c of v **do**
7: $\quad\quad$ **if** the containing query of v_c has $\kappa == \kappa_{max}$ **then**
8: $\quad\quad\quad f \leftarrow isFilter(v_c)$? monitored selectivity of v_c : 1
9: $\quad\quad\quad sumOfSel \mathrel{+}= f$
10: $\quad$ **if** *sumOfSel* < 1 **then**
11: $\quad\quad$ Remove κ satisfying $\kappa == \kappa_{max}$ from $\mathcal{K}(v)$
12: $\quad$ **else**
13: $\quad\quad$ *localBufSize* $\leftarrow \kappa_{max}$
14: $\quad\quad$ **break**
15: **if** *localBufSize* > 0 **then**
16: $\quad$ Place a K-slack chain of length $|\mathcal{K}(v)|$ right below v, where $|\mathcal{K}(v)|$ is the number of remaining κ values in $\mathcal{K}(v)$; and determine the size and the output targets of each K-slack buffer in the chain
17: **for** each child v_c of v whose containing query Q_j has $\kappa_j >$ *localBufSize* **do**
18: $\quad$ Place a K-slack buffer of size ($\kappa_j -$ *localBufSize*) right below v_c

However, looking at the four equations closely, the following relations can be observed:

$$\begin{aligned} mem(C_2^{(v)}) &= r^{(v)} \cdot (\kappa_1 + \kappa_2 - \kappa_2 + f_2 \cdot (\kappa_2 - \kappa_1)) \\ &= r^{(v)} \cdot (\kappa_2 + (f_2 - 1) \cdot (\kappa_2 - \kappa_1)) \\ &= mem(C_1^{(v)}) + r^{(v)} \cdot (f_2 - 1) \cdot (\kappa_2 - \kappa_1) \\ mem(C_3^{(v)}) &= mem(C_2^{(v)}) + r^{(v)} \cdot (f_2 + f_1 - 1) \cdot (\kappa_1 - \kappa_3) \\ mem(C_4^{(v)}) &= mem(C_3^{(v)}) + r^{(v)} \cdot (f_2 + f_1 + f_3 - 1) \cdot \kappa_3 \end{aligned}$$

Because it is assumed that $0 < \kappa_3 < \kappa_1 < \kappa_2$, one can further derive that $mem(C_2^{(v)}) < mem(C_1^{(v)})$ holds if $f_2 < 1$; $mem(C_3^{(v)}) < mem(C_2^{(v)}) < mem(C_1^{(v)})$ holds if $f_2 + f_1 < 1$; and $mem(C_4^{(v)}) < mem(C_3^{(v)}) < mem(C_2^{(v)}) < mem(C_1^{(v)})$ holds if $f_2 + f_1 + f_3 < 1$.

These observations suggest that a more efficient method can be used to find the memory-optimal candidate configuration for a branch operator v, which is based on only the selectivities of the filter children of v. Algorithm 5.1 describes the main idea of this method, which consists of two parts. In the first part, the algorithm determines to which extent a full K-slack chain placed right below a branch operator v can be pushed down below the filter children of v (lines 1–14). The general idea is as follows: Starting from the candidate local K-slack configuration where a full K-slack chain is placed right below v (e.g., $C_1^{(v)}$ in Figure 5.4b), the algorithm first examines the selectivities of the children of v that are the output targets of the tail

buffer (i.e., $\mathcal{O}_{tgt}(B_3)$ in $C_1^{(v)}$). Any non-filter child of v is assumed to have a selectivity of one. A variable *sumOfSel* is used to track the sum of the selectivities of the children of v that have been examined. If $sumOfSel < 1$, then it means that pushing the tail buffer in the K-slack chain below the examined children has a lower memory cost (e.g., $C_2^{(v)}$ in Figure 5.4c). The algorithm then continues to check, by examining the selectivities of the output targets of the last but one buffer in the full K-slack chain, whether pushing down that buffer as well can further reduce the memory cost. This procedure continues until $sumOfSel > 1$ or all children of v have been examined. If the sum of the selectivities of all children of v is smaller than one, then the candidate local K-slack configuration where one K-slack buffer is placed right below each filter child of v is the optimal local K-slack configuration (e.g., $C_4^{(v)}$ in Figure 5.4e), and will be chosen by the algorithm. In the second part (lines 15–18), based on the buffer push-down decisions made in the first part, the algorithm further determines the size and the output targets of each buffer in the chosen candidate local K-slack configuration.

5.4.2 Solution for a Subplan

This section moves on to discuss the solution for finding the memory-optimal K-slack configuration for an entire subplan G^i that may contain hierarchical branch operators. Namely, the child of a branch operator may be a branch operator as well. To distinguish, a branch operator v is called a *bottom branch operator* if none of its children is again a branch operator; otherwise, v is called a *non-bottom branch operator*. The example subplan G^3 in Figure 5.1 is used in this section to illustrate the proposed solution.

Complexity. Recall from Section 5.4.1 that for a branch operator v, there are at most $N_F(v)+1$ candidate local K-slack configurations for v, where $N_F(v)$ is the number of filter children of v. For a subplan G^i, the set of candidate K-slack configurations for G^i includes all possible combinations of the local candidate configurations of the branch operators in G^i. Hence, there are up to $\prod_1^p(N_F(v_p)+1)$ candidate K-slack configurations for G^i, where p represents the number of branch operators in G^i. For example, for the subplan G^3 in Figure 5.1, the total number of candidate K-slack configurations is up to $(N_F(S_3)+1)\cdot(N_F(F_7)+1) = 3\cdot 3 = 9$.

Without loss of generality, let us assume that the optimal QDDH buffer sizes of the queries contained in G^3 satisfy $0 < \kappa_5 < \kappa_3 < \kappa_4$. Figure 5.5 shows all nine candidate K-slack configurations for G^3 under this assumed relation between κ_3, κ_4, and κ_5. In each row of Figure 5.5, the local K-slack configuration for the filter F_7 is fixed; and in each column, the local K-slack configuration for the source operator S_3 is fixed. The configuration C_9^3 is indeed doing unshared disorder handling.

The discussion of the complexity above implies that it is expensive to search for the optimal K-slack configuration for a subplan G^i by enumerating all candidate configurations to compare their memory costs, especially when there is a large number of candidate configurations. To achieve scalability, in the following, a greedy algorithm, *GREEDY*, and an optimal algorithm, *OPT*, are proposed. The algorithm *GREEDY* trades the memory-optimality of the chosen configuration for low computational cost, and does not perform exhaustive enumeration of all candidate configurations. By paying a bit more computational cost than the algorithm *GREEDY*, the

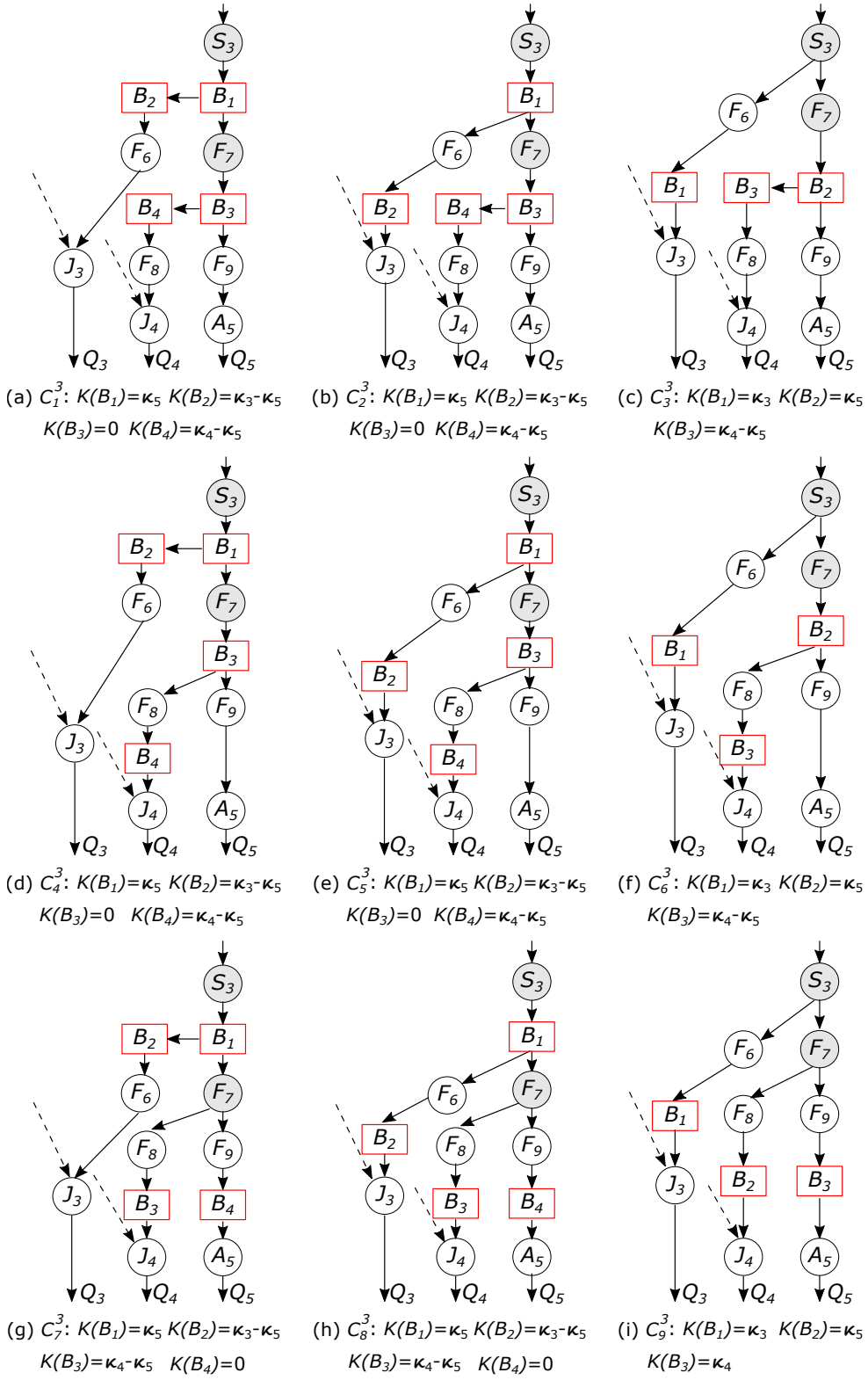


Figure 5.5: All candidate K-slack configurations for the subplan G^3 of the global query plan in Figure 5.1. Assume that the optimal QDDH buffer sizes of the queries in G^3 satisfy $0 < \kappa_5 < \kappa_3 < \kappa_4$.

algorithm *OPT* can find the memory-optimal K-slack configuration for a subplan G^i, yet without enumerating all candidate configurations either.

Both algorithms make use of the notion of *the largest sharable K-slack buffer size for an operator v*, denoted by $\kappa^s(v)$. Generally speaking, $\kappa^s(v)$ represents the largest K-slack buffer size that can potentially be shared among the queries $\mathcal{Q}(v)$ that share an operator v, without violating the latency-minimization condition defined in Section 5.1. Formally, $\kappa^s(v)$ is defined as the smallest optimal QDDH buffer size κ among the queries in $\mathcal{Q}(v)$, i.e., $\kappa^s(v) = min\{\kappa_j | Q_j \in \mathcal{Q}(v)\}$.

Algorithm *GREEDY*

The proposed greedy algorithm—*GREEDY*—determines the K-slack configuration for a subplan G^i based on the following observation: For each branch operator v in G^i, with all other branch operators having their local K-slack configurations fixed, then relations that are similar to those described in Section 5.4.1 can be found between the memory costs of the candidate K-slack configurations for G^i that differ only in the local K-slack configuration for v. For example, considering the configurations in Figure 5.5 column-wise, one can derive the following relations, where r_3 represents the average tuple arrival rate of the stream S_3:

$$
\begin{aligned}
mem(C_7^3) &= mem(C_4^3) + r_3 \cdot f_7 \cdot (f_8 + f_9 - 1) \cdot (\kappa_5 - \kappa_5) \\
mem(C_4^3) &= mem(C_1^3) + r_3 \cdot f_7 \cdot (f_8 - 1) \cdot (\kappa_4 - \kappa_5) \\
mem(C_8^3) &= mem(C_5^3) + r_3 \cdot f_7 \cdot (f_8 + f_9 - 1) \cdot (\kappa_5 - \kappa_5) \\
mem(C_5^3) &= mem(C_2^3) + r_3 \cdot f_7 \cdot (f_8 - 1) \cdot (\kappa_4 - \kappa_5) \\
mem(C_9^3) &= mem(C_6^3) + r_3 \cdot f_7 \cdot (f_8 + f_9 - 1) \cdot \kappa_5 \\
mem(C_6^3) &= mem(C_3^3) + r_3 \cdot f_7 \cdot (f_8 - 1) \cdot (\kappa_4 - \kappa_5)
\end{aligned}
$$

In this specific example, $mem(C_7^3) = mem(C_4^3)$ and $mem(C_8^3) = mem(C_5^3)$ hold regardless of the specific values of the selectivities f_8 and f_9. However, for a different ordering relation between κ_3, κ_4, and κ_5, e.g., when κ_3 is the smallest one, then the relation between $mem(C_7^3)$ and $mem(C_4^3)$, and the relation between $mem(C_8^3)$ and $mem(C_5^3)$ would still depend on the values of f_8 and f_9. Hence, in general, if $f_8 + f_9 < 1$, then the memory-optimal K-slack configuration is guaranteed to be within the third row of Figure 5.5; if $f_8 + f_9 \geq 1$ and $f_8 < 1$, then the optimal configuration is guaranteed to be within the second row of Figure 5.5; if $f_8 = 1$, then the memory costs of the top two candidate configurations in each column of Figure 5.5 are identical, and the optimal configuration can be found within either the first or the second row of Figure 5.5.

Furthermore, one can derive the following relations when considering the configurations in Figure 5.5 row-wise:

$$
\begin{aligned}
mem(C_2^3) &= mem(C_1^3) + r_3 \cdot (f_6 - 1) \cdot (\kappa_3 - \kappa_5) \\
mem(C_3^3) &= mem(C_2^3) + r_3 \cdot (f_6 + f_7 - 1) \cdot \kappa_5 \\
mem(C_5^3) &= mem(C_4^3) + r_3 \cdot (f_6 - 1) \cdot (\kappa_3 - \kappa_5) \\
mem(C_6^3) &= mem(C_5^3) + r_3 \cdot (f_6 + f_7 - 1) \cdot \kappa_5 \\
mem(C_8^3) &= mem(C_7^3) + r_3 \cdot (f_6 - 1) \cdot (\kappa_3 - \kappa_5)
\end{aligned}
$$

Algorithm 5.2 *GREEDY*: a greedy algorithm for determining the K-slack configuration for a subplan G^i that roots at a source operator S_i.

Procedure $GREEDY(G^i)$:
1: $ProcessOp(S_i, 0)$

Procedure $ProcessOp(v, totalBufSizeAboveOp)$:
2: **if** $isLeaf(v)$ **then**
3: **return**
4: $sumOfSel \leftarrow 0$
5: $localBufSize \leftarrow 0$ ▷ the summed size of K-slack buffers placed right below v
6: $\mathcal{K}^s(v) \leftarrow$ distinct values in $\{\kappa^s(v_c) | v_c$ is a child of $v\}$ ▷ recall that $\kappa^s(v_c) = min\{\kappa_j | Q_j \in \mathcal{Q}(v_c)\}$
7: **while** $\mathcal{K}^s(v) \neq \emptyset$ **do**
8: $\kappa^s_{max} \leftarrow max\{\kappa^s | \kappa^s \in \mathcal{K}^s(v)\}$
9: **for** each child v_c of v that satisfies $\kappa^s(v_c) == \kappa^s_{max}$ **do**
10: $f \leftarrow isFilter(v_c)$? monitored selectivity of v_c : 1
11: $sumOfSel \mathrel{+}= f$
12: **if** $SumOfSel < 1$ **then**
13: Remove κ^s satisfying $\kappa^s == \kappa^s_{max}$ from $\mathcal{K}^s(v)$
14: **else**
15: $localBufSize \leftarrow \kappa^s_{max} - totalBufSizeAboveOp$
16: **break**
17: **if** $localBufSize > 0$ **then**
18: Place a K-slack chain of length $|\mathcal{K}^s(v)|$ below v, whose accumulated size is $localBufSize$; determine the size and output targets of each buffer in the chain.
19: **for** each child v_c of v **do** ▷ depth-first traversal
20: **if** $\kappa^s(v_c) < (totalBufSizeAboveOp + localBufSize)$ **then**
21: $ProcessOp\,(v_c, \kappa^s(v_c))$
22: **else**
23: $ProcessOp\,(v_c, (totalBufSizeAboveOp + localBufSize))$

$$mem(C^3_9) = mem(C^3_8) + r_3 \cdot (f_6 + f_7 \cdot (f_8 + f_9) - 1) \cdot \kappa_5$$

These relations imply that no matter which row, i.e., which local K-slack configuration for the operator S_3, was chosen, (1) when $f_6 < 1$, then the memory cost of the second configuration in the chosen row is always lower than the memory cost of the first configuration in that row; and (2) when $f_6 + f_7 < 1$, then the memory cost of the third configuration in the chosen row is always lower than the memory costs of the first two configurations in that row. Observation (2) holds for configurations in the third row of Figure 5.5 as well; because if the third row is chosen, it means that the selectivities of F_8 and F_9 satisfy $f_8 + f_9 < 1$, and therefore $f_6 + f_7 \cdot (f_8 + f_9) < 1$ holds when $f_6 + f_7 < 1$.

Based on the above observations, the algorithm *GREEDY* treats the bottom and the non-bottom branch operators in a subplan G^i equally, and determines the K-slack configuration for G^i by choosing the locally-optimal K-slack configuration for each

branch operator in isolation using Algorithm 5.1. The pseudo-code of *GREEDY* is given in Algorithm 5.2.

Note that with the algorithm *GREEDY*, the placement decisions of K-slack buffers within a subplan G^i do not rely on the order in which the branch operators in G^i are enumerated. However, to determine the size of each K-slack buffer placed along a path from a source operator S_i to a leaf operator in G^i, one needs to consider all K-slack buffers along this path in a top-down manner. Hence, *GREEDY* determines the local K-slack configurations for all branch operators by traversing the subplan G^i in a depth-first way, so that the placement decisions and the properties (i.e., the size and the output targets) of each placed K-slack buffer can be determined in the course of a single traversal of G^i. This depth-first traversal is implemented by recursively invoking the procedure *processOp* (lines 19–23). The optimal global K-slack configuration C^{glob} for a global query plan G^{glob} can be obtained by applying *GREEDY* to each subplan G^i of G^{glob}.

Algorithm OPT

The algorithm *GREEDY* does not guarantee that the produced K-slack configuration for a subplan G^i is memory-optimal. For example, for the subplan G^3 in Figure 5.5, when $f_8 + f_9 < 1$ holds, $f_6 + f_7 \cdot (f_8 + f_9) < 1$ could hold for certain values of f_6 and f_7 that satisfy $f_6 < 1$ and $f_6 + f_7 \geq 1$. However, *GREEDY* would still choose the configuration C_8^3 in Figure 5.5h in that case, because it determines the local K-slack configuration for the branch operator S_3 based on only the selectivities of the filters F_6 and F_7.

The above example implies that to find the memory-optimal K-slack configuration for a subplan G^i, in certain situations, for a branch operator v, it may be necessary to consider the selectivities of both the children and the grandchildren of v. Such a situation occurs when v is a non-bottom branch operator and v has at least one such child v_c that v_c is also a branch operator and no K-slack buffers are placed right below v_c. The branch operator S_3 in configurations C_7^3–C_9^3 (Figure 5.5g–5.5i) is such an example. In this situation, to find the globally-optimal K-slack configuration for the branch operator v, the selectivities of both the children v_c of v and the children of v_c should be considered.

Based on the above observation, the algorithm *OPT* is proposed. The pseudo-code of *OPT* is given in Algorithm 5.3. To find the optimal K-slack configuration for a subplan G^i, the algorithm *OPT* determines the local K-slack configuration for branch operators in G^i in a bottom-up way, so that the situations as described above can be detected and handled properly. However, the size of each K-slack buffer placed in G^i needs to be determined in a top-down way. Hence, compared with the algorithm *GREEDY*, the algorithm *OPT* needs to separate the step of determining the placement of K-slack buffers (lines 1–4) from the step of determining the size of each placed buffer (line 5). Therefore, the computational cost of *OPT* is higher than that of *GREEDY*.

To determine where to place K-slack buffers in a subplan G^i, the algorithm *OPT* uses the notion of *coalesced selectivity*, f_{cs}, for the filters in G^i, and determines the local K-slack configuration for a branch operator v based on the coalesced selectivities of the children of v (lines 6–24). Specifically, the coalesced selectivity f_{cs} of a filter F is defined based on the type of the filter F as follows:

Algorithm 5.3 *OPT*: algorithm for determining the optimal K-slack configuration for a subplan G^i that roots at a source operator S_i.

Procedure $OPT(G^i)$:
1: $nonLeafList \leftarrow$ list of all non-leaf operators in G^i, which are collected by traversing G^i in depth-first post-order
2: **while** $nonLeafList \neq \emptyset$ **do**
3: $v \leftarrow nonLeafList.pop()$
4: $DetermineBufPlacementAtOp(v)$
5: $DetermineBufSizeAtOp(S_i, 0)$

Procedure $DetermineBufPlacementAtOp(v)$:
6: $sumOfCoalescedSel \leftarrow 0$
7: $\mathcal{K}^s(v) \leftarrow$ distinct values in $\{\kappa^s(v_c) | v_c$ is a child of $v\}$ ▷ recall that $\kappa^s(v_c) = min\{\kappa_j | Q_j \in \mathcal{Q}(v_c)\}$
8: **while** $\mathcal{K}^s \neq \emptyset$ **do**
9: $\kappa^s_{max} \leftarrow max\{\kappa^s | \kappa^s \in \mathcal{K}^s(v)\}$
10: **for** each child v_c of v that satisfies $\kappa^s(v_c) == \kappa^s_{max}$ **do**
11: $f \leftarrow 1$
12: **if** $isFilter(v_c)$ **then**
13: $f \leftarrow isBranchOp(v_c)$? $v_c.f_{cs}$: monitored sel. of v_c
14: $sumOfCoalescedSel \mathrel{+}= f$
15: **if** $sumOfCoalescedSel < 1$ **then**
16: Remove κ^s satisfying $\kappa^s == \kappa^s_{max}$ from $\mathcal{K}^s(v)$
17: **else**
18: $v.hasBuf \leftarrow$ true
19: Annotate v with the current $\mathcal{K}^s(v)$ ▷ the annotation is for calculating the sizes of placed buffers later on
20: **break**
21: **if** $isFilter(v)$ **then** ▷ determine the coalesced selectivity of v
22: $v.f_{cs} \leftarrow$ monitored selectivity of v
23: **if** $isBranchOp(v)$ **and** $v.hasBuf == false$ **then**
24: $v.f_{cs} \leftarrow v.f_{cs} \cdot sumOfCoalescedSel$

Procedure $determineBufSizeAtOp(v, totalBufSizeAboveOp)$:
25: **if** $v.hasBuf == true$ **then** ▷ at lease one buffer is placed right below v
26: $\mathcal{K}^s(v)' \leftarrow \mathcal{K}^s(v)$ annotated with v
27: $\kappa^s_{max} \leftarrow max\{\kappa^s | \kappa^s \in \mathcal{K}^s(v)'\}$
28: $localBufSize \leftarrow \kappa^s_{max} - totalBufSizeAboveOp$
29: **if** $localBufSize > 0$ **then**
30: Place a K-slack chain of length $|\mathcal{K}^s(v)'|$ below v, whose accumulated size is $localBufSize$; determine the size and output targets of each buffer in the chain.
31: **for** each child v_c of v **do**
32: **if** $\kappa^s(v_c) < (totalBufSizeAboveOp + localBufSize)$ **then**
33: $DetermineBufSizeAtOp\,(v_c, \kappa^s(v_c))$
34: **else**
35: $DetermineBufSizeAtOp(v_c, (totalBufSizeAboveOp + localBufSize))$

- F is not a branch operator: the coalesced selectivity f_{cs} of F is equal to the selectivity f of F, which is monitored at the query runtime.
- F is a branch operator:
  - If it is determined that a K-slack chain consisting of at least one K-slack buffer is placed right below F, then the coalesced selectivity f_{cs} of F is equal to the monitored selectivity of F.
  - If it is determined that no K-slack buffer is placed right below F, then the coalesced selectivity f_{cs} of F is equal to the product between the monitored selectivity of F and the sum of the coalesced selectivities of the children of F. A non-filter child of F is assumed to have a selectivity of one. For example, the coalesced selectivity f_{cs} of the filter F_7 in C_7^3 to C_9^3 (Figure 5.5g–5.5i) is $f_7 \cdot (f_8 + f_9)$.

The placement decisions of K-slack buffers are marked in the subplan G^i by annotating each operator with a Boolean flag *hasBuf*, whose default value is *false*. When setting the *hasBuf* flag of an operator v to *true*, it means that a K-slack chain consisting of at least one K-slack buffer is placed right below the operator v. After the placement decisions have been made, the algorithm *OPT* determines the size of each placed K-slack buffer in the same way as the algorithm *GREEDY* does (lines 25–35 of Algorithm 5.3). Note that even if the flag *hasBuf* of an operator v is set to *true*, the total K-slack buffer size right below v may turn out to be zero because of the buffers placed above v. For example, the size of the buffer B_4 in the configurations C_7^3 (Figure 5.5g) and C_8^3 (Figure 5.5h) is indeed zero.

5.5 Runtime Adaptation

To achieve the objective of QDDH, in the course of the query processing, the *Global K-slack Configuration Calculator* in the QDDH-framework instantiation in Figure 5.2 must react to the time-varying data and disorder characteristics within the input streams, and dynamically adapt the global K-slack configuration for a global query plan G^{glob}. In the following, Section 5.5.1 introduces different strategies for triggering the adaptation of the global K-slack configuration at the query runtime. Section 5.5.2 shows how careless adaptations may cause incorrect processing semantics when doing shared disorder handling, and presents solutions which can provide semantics-preserving adaptations.

5.5.1 Strategies for Triggering Adaptations

When doing unshared disorder handling for queries in a global query plan G^{glob}, each K-slack buffer B placed in G^{glob} is used only by one query (cf. C_9^3 in Figure 5.5i). The runtime adaptation can be done at a very low cost in terms of the CPU time, because it involves changing only the sizes $K(B)$ of the placed buffers. In contrast, when doing shared disorder handling, the runtime adaptation often involves K-slack chains and requires changing both the sizes and the output targets of the buffers in an old K-slack chain. Such an adaptation often requires additional effort to preserve the correct processing semantics and hence introduces additional overhead.

Intuitively, when doing shared disorder handling, the more often that the adaptations are triggered, the more likely that the optimal K-slack configuration is applied at any point in time, and the higher the overall adaptation cost.

The proposed instantiation of the QDDH framework for concurrent queries implements the following adaptation-triggering strategies for shared disorder handling. Note that when a new optimal QDDH buffer size κ for a query Q in a global query plan G^{glob} is derived, only the K-slack configuration of each subplan G^i, whose root S_i is a source operator involved in the query Q, needs to be adapted.

- *Eager adaptation*: Trigger an adaptation for each involved subplan G^i when a new and different optimal QDDH buffer size is derived for a query Q.
- *Lazy adaptation*: For each subplan G^i in G^{glob}, trigger an adaptation when each query contained in G^i has reported a new optimal QDDH buffer size since the last adaptation for G^i.
- *λ-lazy adaptation*: For each subplan G^i in G^{glob}, trigger an adaptation when a fraction λ, $\lambda \in (0,1)$, of the queries contained in G^i have reported a new optimal QDDH buffer size since the last adaptation for G^i; λ is a configurable parameter.

These strategies will be compared experimentally in Section 5.6. In addition, because of the adaptation cost, adapting the K-slack configuration may cancel off the benefit of applying a new configuration. Hence, each adaptation strategy also needs to avoid such profitless adaptations. This will be discussed in detail in Section 5.5.2.

5.5.2 Semantics-Preserving Adaptations

Adapting the K-slack configuration for a subplan G^i consists of a set of sub-tasks, each of which is to adapt the local K-slack configuration for a specific operator in G^i. When doing shared disorder handling, adapting the local K-slack configuration of a branch operator without caution may lead to incorrect processing semantics. This is illustrated in Figure 5.6. Let us consider a branch operator v_1 that has three children v_2, v_3, and v_4. Assume that the current local K-slack configuration for the branch operator v_1 is as shown in Figure 5.6a. Assume also that v_1 has received four tuples e^{ts} ($ts = 1, 2, 3, 4$), where ts represents the timestamp of a tuple. Figure 5.6a shows the content of each K-slack buffer in the current local K-slack configuration, as well as the tuples that have been sent to each child of v_1.

Figure 5.6b–5.6d show three possible new local K-slack configurations for the branch operator v_1. Take the configuration $C_1^{(v_1)}$ in Figure 5.6b as an example. The length of the K-slack chain and the size of each buffer in the K-slack chain in $C_1^{(v_1)}$ are the same as those in the configuration in Figure 5.6a; but the output targets of the K-slack buffers in $C_1^{(v_1)}$ are different from the output targets of the K-slack buffers in the configuration in Figure 5.6a. When changing from the local K-slack configuration in Figure 5.6a to $C_1^{(v_1)}$ in Figure 5.6b, to avoid incorrect processing semantics, the system must prevent releasing the tuple e^2 to the child v_3 once again, and prevent missing releasing the tuple e^2 to the child v_4. The former problem is termed as *duplicate release* and the latter is termed as *missing release*.

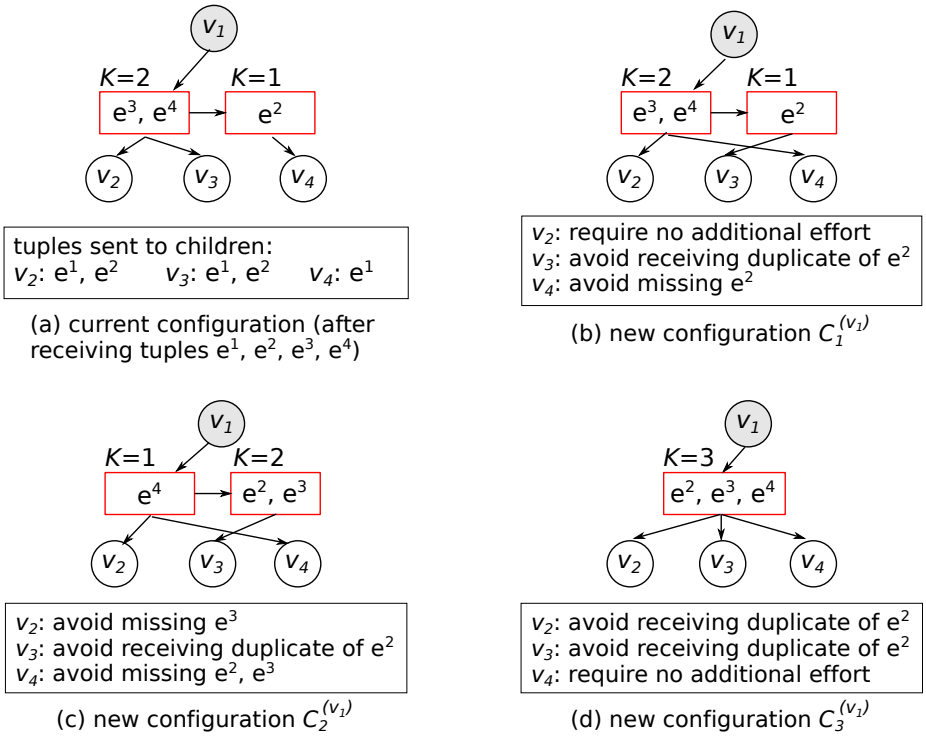


Figure 5.6: Adaptation of the local K-slack configuration for a branch operator.

In general, to support semantics-preserving adaptations of the local K-slack configuration for a branch operator v, the system needs to determine, for each child v_c of v, whether the adaptation is semantically safe for v_c, or additional effort is needed to avoid the *duplicate-release* problem or the *missing-release* problem. Let $K_a(v_c)$ denote the accumulated buffer size between the branch operator v and a child v_c of v, i.e., $K_a(v_c) = \sum_{l=1}^{p} K(B_l)$, where B_l represents the l-th buffer in the K-slack chain placed right below v, and the K-slack chain has $v_c \in \mathcal{O}_{tgt}(B_p)$. The situation in which each child v_c of a branch operator v will be when adapting to a new local K-slack configuration for v can be determined by comparing the accumulated buffer size $K_a(v_c)$ between v and v_c in the current configuration with that in the new configuration, denoted by $K'_a(v_c)$. Specifically, the adaptation is semantically safe for v_c if $K'_a(v_c) == K_a(v_c)$; the *missing-release* problem needs to be avoided if $K'_a(v_c) < K_a(v_c)$; and the *duplicate-release* problem needs to be avoided if $K'_a(v_c) > K_a(v_c)$.

The Basic Method

The naive way to perform semantics-preserving adaptations for a branch operator v is to build a new K-slack chain based on the new local K-slack configuration determined for v, and then migrate tuples from the current K-slack chain to the new K-slack chain. The tail buffer of the current K-slack chain can always be reused when building the new K-slack chain, and only the tuples in the other buffers of the current K-slack chain need to be migrated. This buffer-reuse strategy is referred to as *naive buffer-reuse* hereafter. In the course of the tuple migration, the *missing-release* problem can be handled naturally. To avoid the *duplicate-release* problem, the branch operator

v can remember the maximum timestamp among the tuples that have been released to each child v_c, denoted by $T(v_c)$. For example, in Figure 5.6a, $T(v_2) = T(v_3) = 2$ and $T(v_4) = 1$. The branch operator can then avoid sending any tuple e that satisfies $e.ts \leq T(v_c)$ to its child v_c again during the tuple migration.

After constructing a new K-slack chain in the way as described above, the *duplicate-release* problem may still occur during the future query processing and must be prevented. However, this problem cannot be solved by comparing only the timestamp of a new output tuple of the branch operator v with $T(v_c)$. To show the reason for this, let us consider again the configuration $C_1^{(v_1)}$ in Figure 5.6b. Assume that a new tuple e' is output by the branch operator v_1, and the timestamp of e' is 2. The tuple e'^2 would travel through the new K-slack chain and stop at the tail buffer in the chain. Note that the tail buffer contains the output tuple e^2 as well, which was output by v_1 earlier. Assume that now a tuple with a timestamp larger than 4 is output by v_1, then both the tuple e^2 and the tuple e'^2 will be emitted from the tail buffer in the K-slack chain in Figure 5.6b. Although the timestamps of both tuples are not larger than $T(v_3) = 2$, the tuple e'^2 is indeed not a duplicate and should be forwarded to the child v_3.

To be able to detect true duplicates in the future query processing, the branch operator v_1 is instrumented to maintain an *adaptation counter*, n_{ac}, whose value is incremented each time the local K-slack configuration for v_1 is updated. Each output tuple of v_1 is annotated with the present value of n_{ac}, and the timestamp-based duplicate-check is applied only for a new tuple e that satisfies $e.n_{ac} < v_1.n_{ac}$.

Migrating tuples from an old K-slack chain to a new one introduces additional latency to query results. Hence, it may cancel off the intended benefit of applying a new K-slack configuration C^i for a subplan G^i, if C^i is triggered by a reduction in the optimal QDDH buffer size κ of a query contained in G^i. To avoid such profitless adaptations of K-slack configurations, a model can be trained to estimate the latency penalty of updating to a new configuration C^i. Let us denote this latency penalty by $LP(C^i)$. A simple method to train such a model is to assume a linear relation between $LP(C^i)$ and the total number of tuples that need to be migrated when adapting to C^i. The new K-slack configuration C^i is applied only if $LP(C^i)$ is smaller than the latency reduction (i.e., buffer-size reduction) intended by C^i.

The Optimized Method

The basic method discussed above reuses only the tail buffer of an old K-slack chain when updating to a new local K-slack configuration for a branch operator. However, with a closer study, one can find that if the new local K-slack configuration for an operator v satisfies the *condition for smart buffer-reuse* that is defined below, then the full K-slack chain in the current local configuration can be reused, though appending new buffers to the tail and updating the sizes and the output targets of the buffers in the chain may be necessary. Reusing the full K-slack chain can help to reduce the adaptation cost. This buffer-reuse strategy is referred to as *smart buffer-reuse* hereafter. For ease of presentation, let $idx_B(v_c)$ denote the index of the buffer in a K-slack chain, whose output targets include the child v_c of a branch operator v. For example, in Figure 5.6a, $idx_B(v_2) = idx_B(v_3) = 1$ and $idx_B(v_4) = 2$. The prime symbol ($'$) is used to indicate a value in the new local K-slack configuration.

Condition for smart buffer-reuse: When updating the current local K-slack configuration for an operator v to a new configuration, the full K-slack chain in the current local K-slack configuration can be reused if there exists no child v_c of v, such that $idx'_B(v_c) < idx_B(v_c)$ and $K'_a(v_c) > K_a(v_c)$. Recall that $K_a(v_c)$ represents the accumulated K-slack buffer size between the operator v and its child v_c.

For the example in Figure 5.6, the configurations $C_1^{(v_1)}$ and $C_2^{(v_1)}$ both satisfy the condition for smart buffer-reuse, whereas the configuration $C_3^{(v_1)}$ does not.

When using the *smart buffer-reuse* strategy, before updating the local *K*-slack configuration for an operator v to a new one, it is first checked whether the new configuration satisfies the condition for smart buffer-reuse. If it does, new buffers are appended to the tail of the current *K*-slack chain if necessary, and the size and the output targets of each buffer in the chain is updated according to the new configuration. In addition, to handle the *missing-release* problem, for each child v_c of v, a copy of the tuples in the buffers indexed from $idx'_B(v_c)+1$ to $idx_B(v_c)$ are sent to the child v_c. The *duplicate-release* problem is handled in the same way as with the *naive buffer-reuse* strategy.

5.6 Evaluation

Same as the two instantiations of the QDDH framework described in Chapter 4, the instantiation for concurrent queries with shared operators proposed in this chapter was implemented in the prototypical version of SAP ESP (cf. Section 4.3). The evaluation of this proposed instantiation consists of two parts. The first part (Section 5.6.2) focuses on comparing the performance of the following four algorithms for determining the *K*-slack configuration for a global query plan G^{glob}: (1) *UNSHARED*, which does unshared disorder handling for all queries contained in G^{glob}, (2) *GREEDY* (cf. Algorithm 5.2), (3) *OPT* (cf. Algorithm 5.3), and (4) *NAIVE*, which searches for the optimal *K*-slack configuration for a subplan G^i in G^{glob} via exhaustive enumeration of all possible candidate *K*-slack configurations. The second part (Section 5.6.3) focuses on studying the runtime overhead of the adaptation-triggering strategies and the buffer-reuse strategies introduced in Section 5.5. Each experiment was run three times and the average result is reported in this section.

5.6.1 Setup

The datasets used in this evaluation were created based on a data stream containing 23-minute soccer data collected from the RTLS introduced in Section 4.3.1. The three global query plans in Figure 5.7 were used in this evaluation, which have increasing number of candidate global *K*-slack configurations. To test with `GQPlan1`, each tuple in the original stream was extended with one more attribute, whose value was picked randomly and uniformly from the integer range $[1, 100]$. To test with `GQPlan2` and `GQPlan3`, the original stream was split into two streams, one for each team in the soccer game. Each tuple was extended with two attributes, whose values were taken randomly and uniformly from the integer range $[1, 100]$ as well. For each designed global query plan, Figure 5.7 shows the predicates of the filters in the global query plan as well as the selectivity of each filter (i.e., the number within parentheses) The

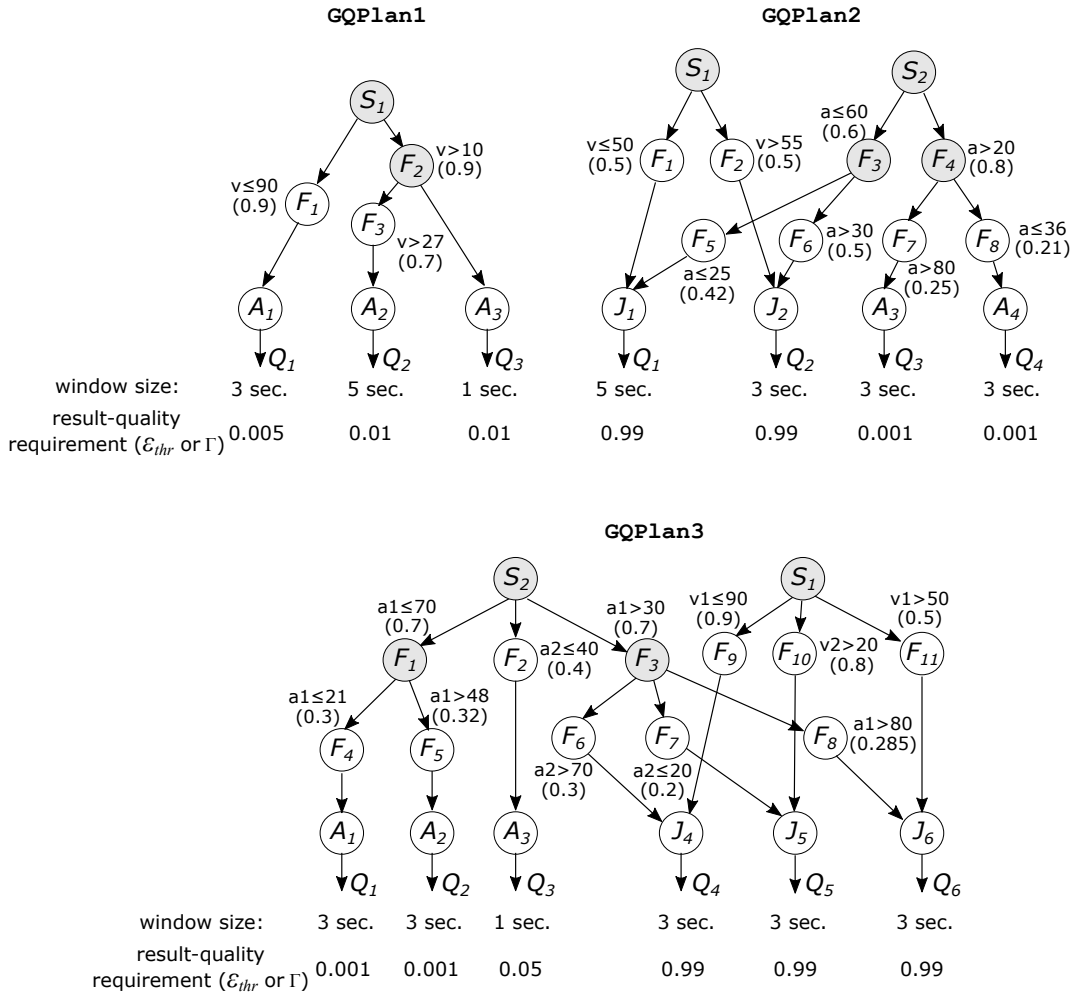


Figure 5.7: Global query plans used to evaluate the instantiation of the QDDH framework for concurrent queries with shared source and filter operators.

window size shown below each query in a global query plan is the size of the sliding window applied over each input stream of the query. For example, for the 2-way join query Q_2 in `GQPlan2`, a sliding window of 3 seconds is applied to each input stream of the operator J_2.

The window slide for each SWA query was set to 0.1 second. Hence, the optimal QDDH buffer size κ of each SWA query was recalculated every 0.1 second (cf. Section 4.1.2). The *Buffer Manager*s in Figure 5.2 for individual MSWJ queries were instantiated with the default parameter configuration listed in Table 4.2. Hence, the optimal QDDH buffer size κ of each MSWJ query was recalculated every 1 second (cf. Section 4.2.2).

Figure 5.7 also shows the result-quality requirement for each query in the three global query plans. The result-quality metrics for SWA queries and MSWJ queries, i.e., the result relative-error threshold (ϵ_{thr}, δ) and the recall requirement Γ, are as defined in Section 4.1.1 and Section 4.2.1, respectively. For SWA queries, only the values of ϵ_{thr} are shown in the figure; the confidence level δ in each relative-error threshold was set to 0.05. For each global query plan, the result-quality metrics for

		Avg. end-to-end latency (sec.)		Req. fulfillment ratio Φ (%)	
		No-K-slack	*Max-K-slack*	*No-K-slack*	*Max-K-slack*
GQPlan1	Q1	0.22	17.26	55.9	99.98
	Q2	0.23	17.28	87.75	100
	Q3	0.22	17.26	53.5	99.93
GQPlan2	Q1	0.27	20.37	10.52	100
	Q2	0.32	17.75	9.99	100
	Q3	0.07	8.23	65.12	99.54
	Q4	0.08	8.88	71	99.29
GQPlan3	Q1	0.06	16.18	60.46	99.3
	Q2	0.06	6.38	61.37	99.61
	Q3	0.04	16.16	94.44	99.93
	Q4	0.09	18.27	6.42	100
	Q5	0.08	20.87	6.95	100
	Q6	0.13	18.26	7.31	100

Table 5.1: Results of the *No-K-slack* and the *Max-K-slack* baseline disorder handling approaches for the global query plans in Figure 5.7.

the queries in the plan were chosen in such a way that the queries have different optimal QDDH buffer sizes.

As described in Section 4.3.1, the performance of QDDH is measured in terms of the average end-to-end latency and the requirement fulfillment ratio of each query (i.e., $\Phi(\epsilon_{thr})$ for a SWA query and $\Phi(\Gamma)$ for an MSWJ query). Again, *No-K-slack* and *Max-K-slack* (cf. Section 4.3.2) were taken as baseline disorder handling approaches. The results of these two baseline approaches are shown in Table 5.1. However, different from Chapter 4, this chapter focuses more on the task of determining the *K*-slack configuration—based on the derived optimal QDDH buffer size κ of each query—to conduct QDDH for a global query plan with shared source and filter operators efficiently with respect to the time and the memory consumption, rather than on the task of deriving the optimal QDDH buffer size κ for an individual query itself.

5.6.2 Performance of Alternative Algorithms for Computing *K*-slack Configurations

In this experiment, the adaptation-triggering strategy was fixed to *eager adaptation* and the buffer-reuse strategy was fixed to *smart buffer-reuse*.

Figure 5.8a shows the average time of computing a global *K*-slack configuration for each global query plan in Figure 5.7, taking the runtime-derived optimal QDDH buffer sizes of the queries contained in the global query plan as input. When doing unshared QDDH, the placement of *K*-slack buffers within a global query plan does not change at the query runtime; hence, Figure 5.8a does not contain the results for the algorithm *UNSHARED*. One can see that (1) the additional computational cost that the algorithm *OPT* has over the algorithm *GREEDY* is negligible, and the computation times of both algorithms are significantly lower than the computation time of the algorithm *NAIVE*, especially for global query plans with a large number

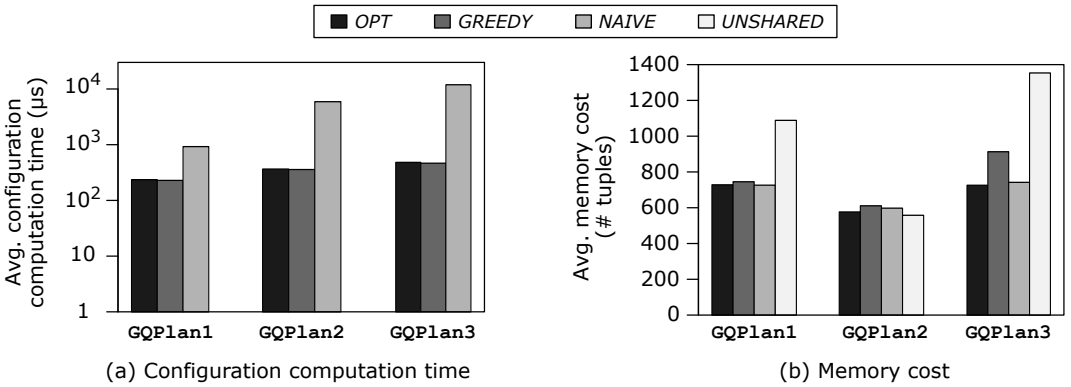
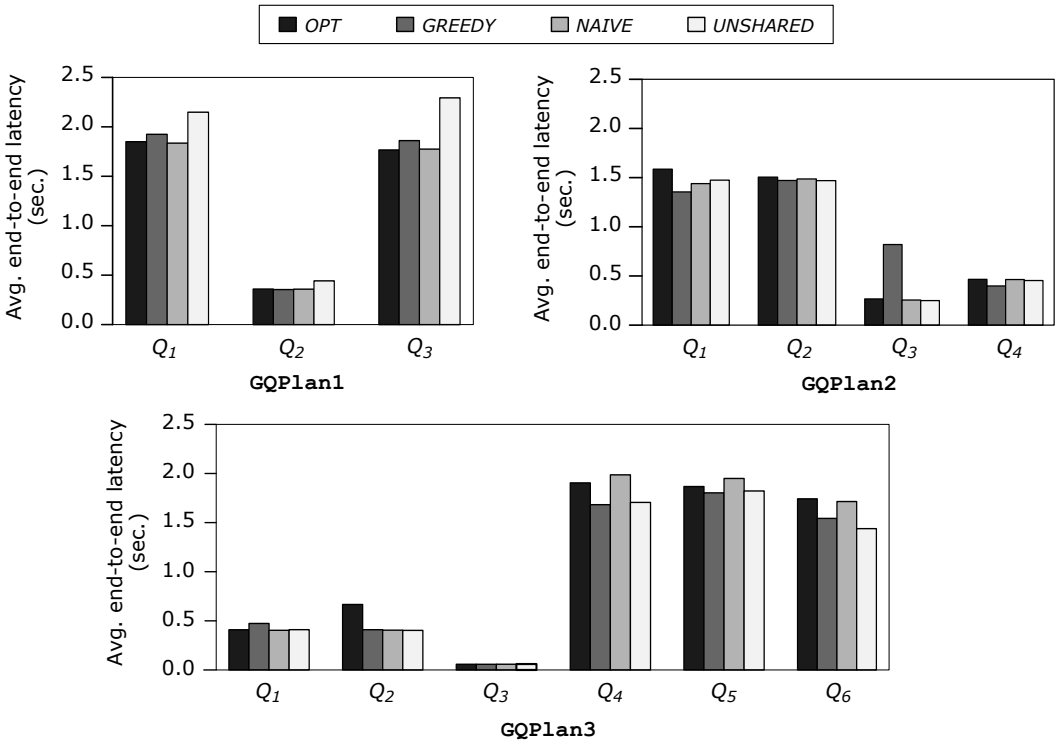


Figure 5.8: The runtime of alternative algorithms for determining the global *K*-slack configuration for the global query plans in Figure 5.7, and the memory costs of the produced global *K*-slack configurations.

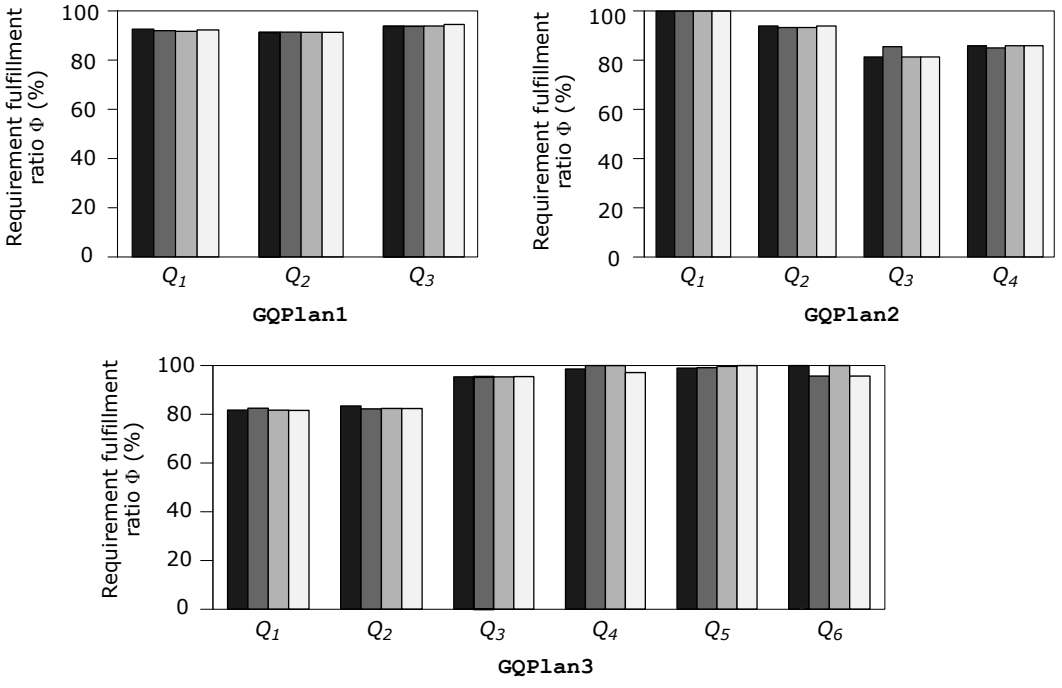
of candidate global *K*-slack configurations (e.g., `GQPlan3`); (2) the computation time of each algorithm increases as the number of candidate global *K*-slack configurations increases.

Figure 5.8b shows for each global query plan the average memory cost of disorder handling over time, i.e., the time-average number of tuples kept by all *K*-slack buffers placed in a global query plan. The three global query plans in Figure 5.7 were designed in such a way that for `GQPlan1` and `GQPlan3`, shared disorder handling is memory-optimal, whereas for `GQPlan2`, unshared disorder handling is memory-optimal. It can be seen that the algorithm *OPT* can always find the memory-optimal global *K*-slack configuration, whereas the algorithm *GREEDY* cannot. For example, for `GQPlan3`, the average memory cost of the configurations produced by *GREEDY* is about 200 tuples higher than the average memory cost of the configurations produced by *OPT*. Note that for `GQPlan3`, where the fan-out degree of a branch operator is at most three, the memory cost of unshared disorder handling is already about twice as high as the memory cost of optimal shared disorder handling. Hence, one can expect that the memory-saving of shared disorder handling would be more significant if the branch operators in the global query plan had a higher fan-out degree.

The average end-to-end latencies of the queries in each global query plan are shown in Figure 5.9a. In theory, for a given global query plan, if the four configuration-computation algorithms compute new *K*-slack configurations at exactly the same points in the course of the query processing, and no new tuples arrive during each configuration computation, then one can expect that, for each query in the global query plan, all four algorithms produce the same average end-to-end latency. However, in reality, the four algorithms have different runtime. Moreover, the prototypical DSPS used in this dissertation adopts a multi-thread implementation, and the configuration computation within one thread does not block the other threads that run query operators. As a result, the four algorithms are rarely triggered at the same points with respect to the query-processing progress, even though the applied adaptation-triggering strategy was the same. Each *K*-slack configuration applied at one point in time influences the subsequent behavior of the QDDH. Hence, different average end-to-end latencies were observed for the same query. However, for each query, the average latency produced by the algorithm *OPT* is at most 0.35 second



(a) Average end-to-end latency produced under QDDH for each query



(b) Result quality in terms of the requirement fufillment ratio Φ produced under QDDH for each query

Figure 5.9: QDDH performance of the global *K*-slack configurations produced by alternative configuration-computation algorithms for the global query plans in Figure 5.7.

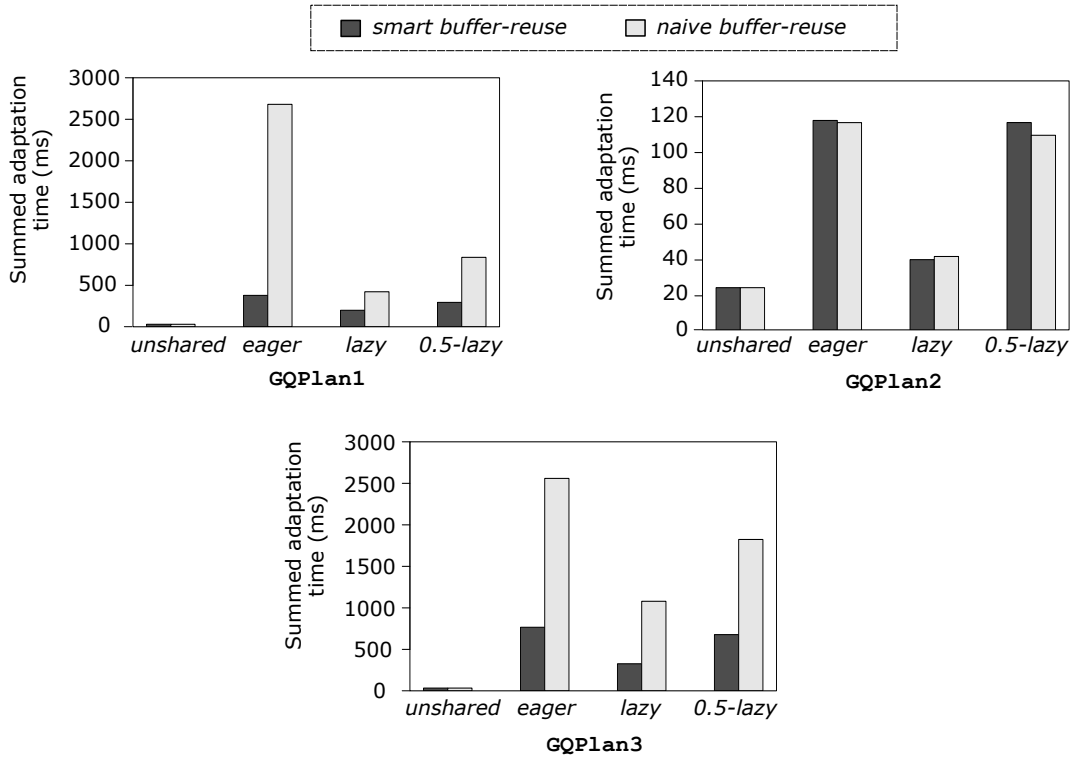


Figure 5.10: Total time consumed by updating to newly-computed global K-slack configurations during the query processing under different combinations of adaptation-triggering strategies and buffer-reuse strategies.

higher than the average latency produced by the algorithm *UNSHARED* (e.g., for Q_6 in `GQPlan3`).

Figure 5.9b shows the query-result quality in terms of the requirement fulfillment ratio Φ (cf. Section 4.3.1) achieved for each query. For a specific query, all four configuration-computation algorithms produced similar result qualities. The achieved requirement fulfillment ratio Φ was at least 80% for SWA queries, and was at least 90% for MSWJ queries. Looking at Figure 5.9 and Table 5.1 together, it can be seen that the proposed QDDH approach achieved a high requirement fulfillment ratio Φ compared with the *No-K-slack* baseline approach, while incurring a much lower end-to-end latency compared with the *Max-K-slack* baseline approach, which again shows the effectiveness of QDDH.

5.6.3 Overhead of Runtime Adaptation

This experiment focused on comparing unshared disorder handling with shared disorder handling with respect to the time needed to update an existing global K-slack configuration to a new one at the query runtime (excluding the configuration-computation time), as well as comparing the three adaptation-triggering strategies—*eager adaptation*, *lazy adaptation*, and *λ-lazy adaptation*—and the two buffer-reuse strategies—*naive buffer-reuse* and *smart buffer-reuse*—that were introduced in Section 5.5. The algorithm *OPT* was used to compute global K-slack configurations for shared disor-

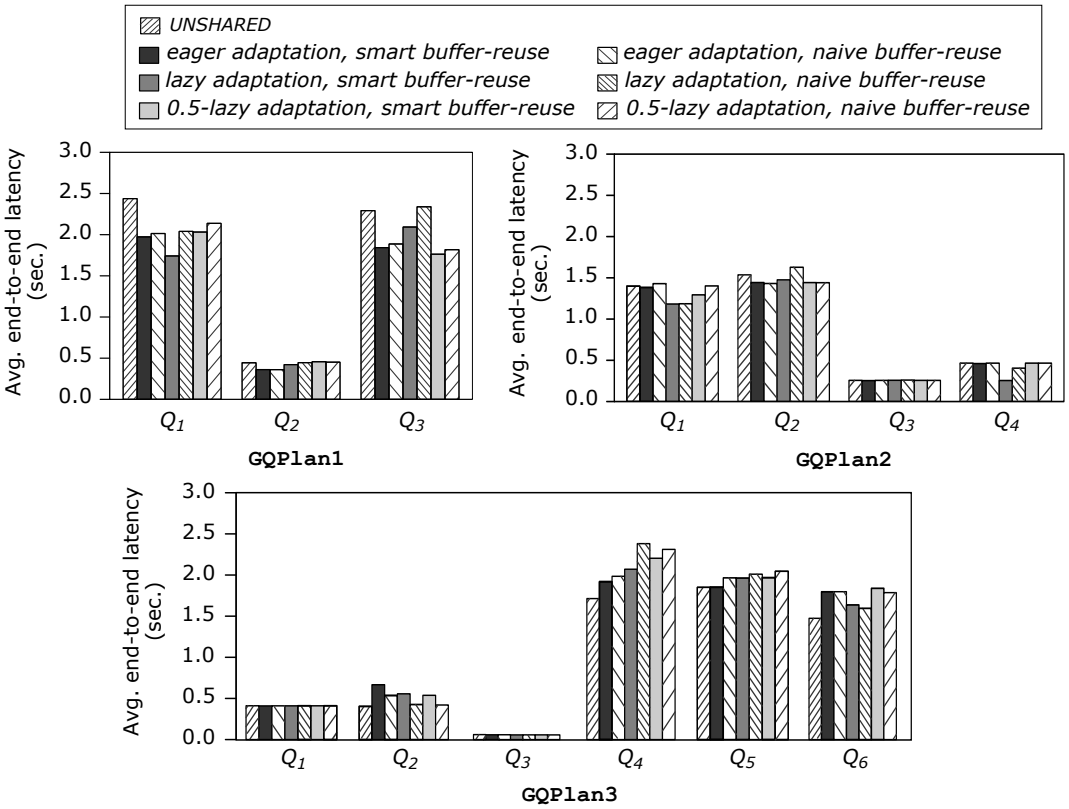
	GQPlan1		GQPlan2		GQPlan3	
	smart reuse	*naive reuse*	*smart reuse*	*naive reuse*	*smart reuse*	*naive reuse*
UNSHARED	0.74		0.79		0.73	
eager adaptation	60	400	36	37	179	657
lazy adaptation	64	132	39	41	295	992
0.5-lazy adaptation	63	155	33	34	171	473

Table 5.2: Average time (μs) needed to update an existing global K-slack configuration to a newly-computed global K-slack configuration during the query processing under different combinations of adaptation-triggering strategies and buffer-reuse strategies.

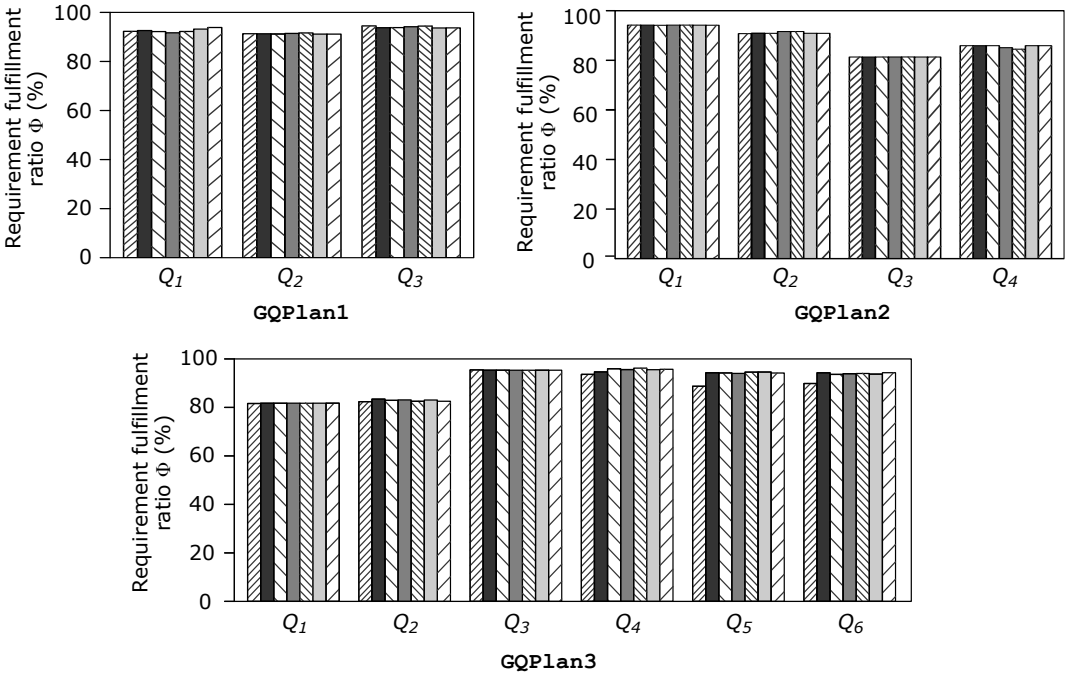
der handling, and the value of λ in the λ-*lazy* adaptation-triggering strategy was set to 0.5.

Figure 5.10 shows the total time consumed by adapting to new K-slack configurations during the processing of a global query plan, and Table 5.2 shows the average time of a single adaptation. One can see that, for each global query plan, unshared disorder handling has much lower adaptation overhead than shared disorder handling. Moreover, because the *naive buffer-reuse* strategy behaves differently from the *smart buffer-reuse* strategy only for K-slack chains of length at least two, and with unshared disorder handling, no such K-slack chains are used, the adaptation times under both buffer-reuse strategies—both the summed adaptation time and the average adaptation time—are the same for unshared disorder handling under each adaptation-triggering strategy. With shared disorder handling, the summed adaptation times of the three adaptation-triggering strategies have the relation of *lazy adaptation* $<$ *0.5-lazy adaptation* $<$ *eager adaptation* for each buffer-reuse strategy; and the summed adaptation time of the *naive buffer-reuse* strategy is higher than that of *smart buffer-reuse* strategy for each adaptation-triggering strategy, especially for the global query plans for which shared disorder handling is preferred by the algorithm *OPT*, i.e., `GQPlan1` and `GQPlan3` (cf. Figure 5.8b). The superiority of *smart buffer-reuse* can be observed from the average time of a single adaptation as well. In general, with the algorithm *OPT* and the *smart buffer-reuse* strategy, computing a new global K-slack configuration (cf. Figure 5.8a) and adapting to the newly-computed configuration can be done within 1 millisecond on average for each global query plan in Figure 5.7.

Figure 5.11a and Figure 5.11b show the average end-to-end latency and the result quality produced for each query, respectively. From Figure 5.11b, one can see that *naive buffer-reuse* in general leads to a higher end-to-end latency than *smart buffer-reuse* under any adaptation-triggering strategy. However, there is no strong correlation between the applied adaptation-triggering strategy and the produced end-to-end latency; because, as shown previously, the adaptation overhead was very small even when the *eager adaptation* strategy was used. All adaptation-triggering strategies and buffer-reuse strategies produced similar requirement fulfillment ratios for a specific query.



(a) Average end-to-end latency produced under QDDH for each query



(b) Result quality in terms of the requirement fufillment ratio Φ produced under QDDH for each query

Figure 5.11: Performance of the instantiation of the QDDH framework for concurrent queries with shared operators, under different combinations of adaptation-triggering strategies and buffer-reuse strategies.

Summary of the Experimental Results In summary, the experimental results presented in this section show that, using the algorithm *OPT* to compute global *K*-slack configurations for a given global query plan G^{glob}, and the smart buffer-reuse strategy to perform adaptations of the global *K*-slack configuration at the query runtime, the proposed instantiation of the QDDH framework for concurrent SWA and MSWJ queries with shared source and filter operators can conduct QDDH for G^{glob} efficiently with respect to both the runtime and the memory consumption, even when the *eager adaptation* strategy is used.

5.7 Related Work

In addition to handling the disorder within data streams, which was discussed in Section 3.5, the work presented in this chapter is also related to *multi-query optimization (MQO)*. Generally speaking, MQO refers to the techniques that optimize the execution of multiple concurrent queries within a system to reduce the total execution cost, thereby improving the system performance. MQO has been studied extensively for database systems since the 1980s [Fin82; Sel88; Roy+00; Zho+07; GAK12; Gia+14], and has been applied in many DSPSs as well, e.g., NiagaraCQ [Che+00] and CACQ [Mad+02]. Particularly, Krishnamurthy et al. [KWF06] proposed methods for efficiently sharing aggregates with different window specifications on the fly. Based on that work, Guirguis et al. [Gui+11] proposed a cost-based optimizer—*weave share*, which exploits the *weaveability* to balance the cost reduced by sharing partial aggregates and the cost incurred at the final aggregation. Later on, Shein et al. [SCL15] introduced methods for calculating the weaveability efficiently. Wang et al. [Wan+06] proposed to share the computation across multiple window-based join queries by slicing a window-based join into a chain of pipelining sliced joins. Hong et al. [Hon+09] proposed a rule-based MQO framework for DSPSs—*RUMOR*, which can integrate new and existing MQO techniques using transformation rules. Seshadri et al. [Ses+09] studied MQO for distributed DSPSs. They proposed to consider the query plan and the deployment of query operators simultaneously, and developed top-down, bottom-up, and hybrid algorithms that exploit operator-level reuse through hierarchical network partitions.

MQO has been studied for MapReduce systems as well. For example, Wang et al. proposed the MRShare framework [Nyk+10], which finds the optimal way of merging MapReduce jobs into groups, and evaluates each group as a single query to improve the execution efficiency. Lei et al. proposed Helix [Lei+15], a MapReduce-based system for shared execution of recurring workloads under user-specified service level agreements (SLAs). Helix employs a SLA-driven optimizer to generate an execution plan for a given set of recurring queries, with the goal of maximizing the overall SLA satisfaction.

The work presented in this chapter is similar to MQO in the sense that it aims to exploit the opportunities of work-sharing as well. However, in contrast to MQO, which focuses on the sharing of query operators, the work in this chapter focuses on the sharing of disorder handling, and is applied on top of MQO.

5.8 Summary

Based on the instantiations of the generic QDDH framework for individual SWA and MSWJ queries that were described in Chapter 4, this chapter presented the instantiation of the QDDH framework for concurrent queries with shared operators, such as source and filter operators. The concept of *K-slack chain* was proposed for sharing the disorder handling of a stream among multiple consumer operators whose containing queries have different user-specified result-quality requirements. Based on the usage of *K*-slack chains, two algorithms—*GREEDY* and *OPT*—were proposed to determine the configuration of *K*-slack buffers within a global query plan to achieve the objective of quality-driven latency minimization. Particularly, the algorithm *OPT* can find the memory-optimal *K*-slack configuration for a given global query plan. To perform time-efficient and semantics-preserving adaptation of the applied *K*-slack configuration at the query runtime, a smart buffer-reuse strategy was proposed, which aims to reuse buffers in an old *K*-slack configuration as much as possible when updating the old *K*-slack configuration to a new one. The experimental results showed the effectiveness of the proposed instantiation of the QDDH framework.

The work presented in this chapter is a first step towards supporting QDDH for concurrent queries. Plenty of opportunities remain to be explored. An interesting direction would be to jointly consider QDDH and the sharing of window-based aggregate or join operators; because, as discussed in Section 5.1, the sharing decisions produced by existing solutions like [KWF06] and [Wan+06] may violate the objective of QDDH.

6

Reducing the Tradeoff via Hybrid Query Execution

Chapter 3–Chapter 5 presented the contribution of this dissertation in the area of providing flexible and user-configurable tradeoff between the performance and the query-result quality when dealing with stream disorder—a representative case of data imperfection in data streams. Recall from Section 1.1 and Section 2.3 that the tradeoff between the performance and the query-result quality can also be caused by the limitations of a DSPS itself. Specifically, when a DSPS has limited processing capacity, it must trade the query-result quality for the performance to avoid system overload. Enhancing the DSPS itself can help reducing, and even avoiding, this tradeoff.

In this chapter, the second research question of this dissertation—"*how to enhance a DSPS to avoid, or reduce to some extent, the performance versus query-result quality tradeoffs caused by system limitations*"—will be studied. The main contributions that this dissertation made in answering this research question include a prototypical, hybrid system which exploits the potential of combining the row-oriented and the column-oriented data layout and processing techniques in data stream processing, as well as the design and implementation of a static cost-based optimizer for optimizing continuous queries executed in such a hybrid system.

After motivating the proposal of such a hybrid system for processing data streams in Section 6.1, Section 6.2 gives an overview of continuous-query execution in the proposed prototype system. Subsequently, Section 6.3 describes the proposed query optimizer in detail. Section 6.4 experimentally studies the effectiveness of the proposed optimizer and Section 6.5 discusses related work.

6.1 Introduction

The majority of existing DSPSs are row oriented, and are able to evaluate window-based operators efficiently when the number of tuples contained in each instantaneous window is small, even with naive operator implementations. Naive implementation means that each result of an operator is computed by scanning the content of the corresponding instantaneous window as many times as necessary. For instantaneous windows with a huge amount of tuples (e.g., several million tuples),

low-latency results can still be guaranteed, by using smart operator implementations. The key ideas behind these smart implementations include calculating query results in a single-pass and incremental way [GKS01; Dob+02; Gha+07; Bha+14; Tan+15], using online-maintained synopsis (e.g., histograms and wavelets) [Cor+12], and utilizing a *divide-and-conquer* approach [Li+05a] (cf. Section 2.3).

However, many stream-based applications also require processing complex aggregate queries [Dob+02], for which these advanced operator-implementation techniques are either not applicable or only able to provide approximate query results. Examples of complex aggregates include quantile and order-statistic computation, correlated aggregate, etc. The key characteristic of complex aggregate queries is that, each query result cannot be computed easily in an incremental way, and usually a full scan of the corresponding instantaneous window is needed. For correlated aggregates, an instantaneous window even needs to be scanned multiple times to compute the exact result for the window. When exact query results are required by an application (e.g., decision support), usually the only operator-implementation option is the naive or near-naive implementation, which typically has a high space and computation cost for large windows. The consequence is a significant increase of the end-to-end latency.

While existing row-oriented DSPSs show limitations in computing exact answers for complex aggregates with large windows, recent studies in the database community have shown that modern database systems are able to process very large datasets with second or even sub-second level latency, by leveraging vertical storage architecture, vectorized query execution, in-memory technology and so on [BKM08; Sik+13]. To demonstrate this, a correlated aggregate query was executed in a state-of-the-art commercial column-oriented in-memory database (CIMDB) and a row-oriented DSPS, and the per-result computation times of this query in these two systems were compared, under increasing per-instantaneous-window tuple-sizes. The query is in the context of stock-market analysis. It calculates the number of companies (`comp`) in each business area (`bs_area`), e.g., retail, banking, insurance, etc., whose market capitalization (`mk_cap`) is greater than 50% of the maximum market capitalization in that area in the past W time units. This scenario can be expressed with relational algebra as follows:

`Comp, bs_area, mk_cap, timestamp:` $R(c, b, m, ts)$
`Data within (T-W, T]:` $S(c, b, m) = \pi_{c,b,m}\sigma_{ts>T-W \wedge ts \leq T}(R)$
`Max. mk_cap in each bs_area:` $M(b, mb) = \pi_{b,b}G_{max(m)}(S)$
`Join between S and M:` $J(c, b, m, mb) = S \bowtie_{S.b=M.b} M$
`Final result:` $C(b, cb) = \pi_{b,b}G_{count(c)}(\sigma_{m>0.5 \cdot mb}(J))$

`T represents the current time.`

The results are shown in Figure 6.1. It can be observed that when the per-result computation time in the DSPS jumped to more than 3.5 seconds for instantaneous windows with 5 million tuples, the CIMDB was able to finish the same computation within 600 milliseconds.

The above results show the potential of column-oriented data layout and processing techniques in processing data streams. However, column-oriented processing is still inferior to row-oriented processing in scenarios where the computation cost of the operators in a query is rather low whereas the tuple-construction operation is

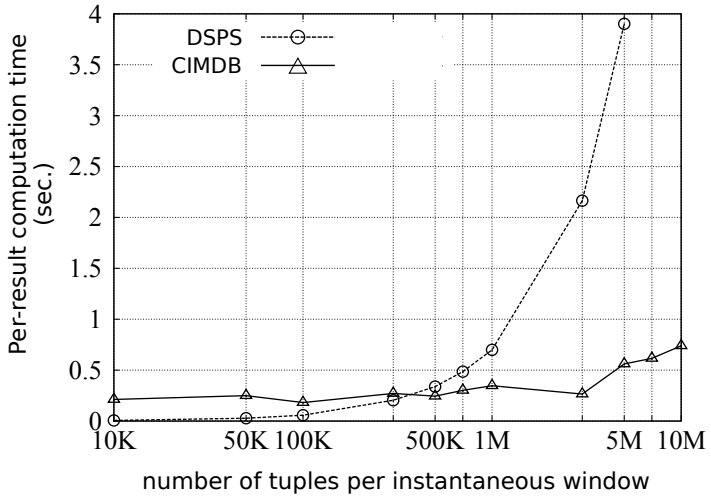


Figure 6.1: Comparison of the computation time of correlated aggregation in state-of-the-art column-oriented in-memory database and row-oriented DSPS

dominant. Motivated by the above observations and following the philosophy of "no one size fits all" [LHB13], this dissertation proposes to combine both the row-oriented and the column-oriented data processing techniques to process continuous queries, aiming to achieve a performance that cannot be matched by either type of processing technique alone. As a proof of concept, a prototype system was built, which consists of a row-oriented DSPS and a CIMDB. Certain fragments of a continuous query are outsourced from the DSPS to the CIMDB, if the outsourcing can lead to higher performance.

One major challenge for such hybrid systems is to find the optimal execution plan for a given continuous query. Existing systems that combine database systems and DSPSs either do not have an optimizer for hybrid query execution at all [Bot+10a], or choose the most suitable system for an entire query [LHB13]. Moreover, none of them has considered the *feasibility* property of execution plans of continuous queries, which describes the capability of an execution plan to keep up with the incoming tuple arrival rate [AN04]. Finally, the heterogeneity between the underlying row-oriented processing engine and the column-oriented processing engine causes the *non-additivity* of the query execution cost [DH02]. Specifically, in our prototype system, the non-additive execution cost means that the cost of executing two consecutive operators in the CIMDB is not necessarily higher than the cost of executing only the first operator in the CIMDB and executing the second operator in the DSPS. This non-additivity of the query-execution cost makes it difficult for a query optimizer to make pruning decisions when enumerating the possible execution plans. Existing solutions used in traditional database systems for handling the cost non-additivity must be extended to consider the feasibility property of execution plans of continuous queries.

To address the above challenge, in this dissertation, a static cost-based optimizer is proposed for optimizing SPJA (select-project-join-aggregate) continuous queries in hybrid systems as proposed in this dissertation. The optimizer fully exploits the potential of hybrid execution of continuous queries across a row-oriented and a col-

umn-oriented data processing engine. For a given continuous query, the optimizer takes into account the feasibility of query execution plans and the non-additivity of the query-execution cost caused by combining two types of data processing engines, and determines the optimal placement for each operator in the query based on the characteristics of the query and the input streams. To reduce the search space of possible query execution plans, the proposed optimizer adopts a two-phase optimization strategy [HS91], which has been used widely in existing federated or parallel database systems, as well as systems with heterogeneous multicore architectures (e.g., [He+09]). In Phase-One of the optimization, an optimal logical query plan is produced; in Phase-Two, placement decisions for all operators in the chosen logical query plan are made. In addition, based on the study of the cost characteristics of the operators placed on the two different data processing engines, the optimizer exploits the opportunities of pruning plans in Phase-Two, to further reduce the search space. The proposed optimization approach was implemented by extending the optimizer of the CIMDB in the prototype system directly.

Note that as pointed by Ayad and Naughton [AN04], static query optimization is a valid approach when the characteristics of the input streams change slowly or the pattern of the changes is predictable, which is often observed in data streams originating from sensors with fixed reporting frequencies. Before moving on to a dynamic optimization solution, it must first be understood what can be achieved by doing static optimization for continuous queries in a hybrid system as proposed in this chapter.

6.2 Hybrid Execution of a Continuous Query

This section gives an overview of the execution of continuous queries in the proposed hybrid system.

The semantics model of data stream processing adopted in this dissertation (cf. Section 2.1) allows a natural semantic mapping between continuous queries and conventional SQL queries, because time-varying relations and R2R operators have straightforward semantic mappings to conventional relations and query operators in database systems, respectively. Without loss of generality, in this chapter, it is assumed that each R2R operator has at most two input relations; and an m-way join with $m \geq 3$ input relations is treated as a sequence of 2-way joins.

Determined by the above semantic mapping between continuous queries and SQL queries, given a logical plan G of a continuous query, the fragments of G that can potentially be migrated from the row-oriented DSPS to the CIMDB are subplans of G that contain only R2R operators (cf. Section 2.1.2). Such a subplan of G is referred to as a *migration candidate*. A composition of several R2R operators produces one relation from one or more relations. Hence, based on the definition of R2R operators in Section 2.1.2, this composition of R2R operators can be regarded as one R2R operator as well. Namely, each migration candidate can be regarded as a *composite R2R operator*. A migration candidate can be translated into a SQL query and executed in the CIMDB. Particularly, the base relations involved in this translated SQL query map to the input relations of the migration candidate; and the result of the SQL query maps to the output relation of the migration candidate.

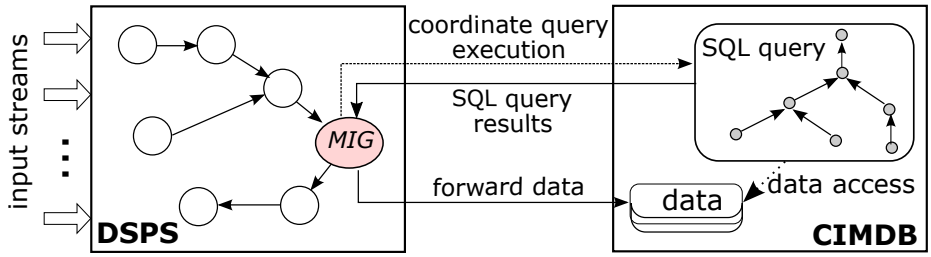


Figure 6.2: Execution of continuous queries in a hybrid system that consists of a row-oriented DSPS and a column-oriented in-memory database (CIMDB).

Figure 6.2 depicts how a continuous query is executed across the DSPS and the CIMDB in the proposed prototype system. The DSPS has several built-in adapters to connect with external data sources, and hence is taken as the gateway of external data streams. The query execution in such a hybrid system involves data transfer between the DSPS and the CIMDB. Specifically, for each migration candidate executed in the CIMDB, relevant input data needs to be transferred from the DSPS to the CIMDB; and results produced by the CIMDB need to be transferred back to the DSPS.

To retain the original query-processing semantics, the SQL query corresponding to a migration candidate must be re-executed in response to changes in the input relations of the migration candidate. To coordinate the data transfer between the two processing engines and the re-execution of the corresponding SQL query, a new operator *MIG* was introduced into the DSPS. A *MIG* operator acts as a wrapper of a migration candidate executed in the CIMDB. It controls the data transfer between the two engines and hides the execution details within the CIMDB from the DSPS. In a parallel processing environment (e.g., a multi-core machine or a distributed deployment of the DSPS), *MIG* operators run in parallel with the other query operators placed in the DSPS. However, from the perspective of the DSPS, each migration candidate wrapped by a *MIG* operator is a black box, and the original pipelined relationships (cf. Section 2.1.3) among the operators within the migration candidate is no longer visible.

Given a logical query plan $G = (\mathcal{V}, \mathcal{ED})$ of a continuous query, let $\mathcal{P}(G) = (\mathcal{V}', \mathcal{ED}', \mathcal{M})$ denote an execution plan of G. Let $v_i^x \in \mathcal{V}'$ represents a physical operator in the execution plan, where $x \in \{dsps, db\}$. Specifically, v_i^{dsps} represents a basic query operator (e.g., selection, join, etc.) placed in the DSPS, and v_i^{db} represents a migration candidate placed in the CIMDB. For ease of reference, in the remainder of this chapter, a basic query operator placed in the DSPS is referred to as a *DSPS-op* and a composite R2R operator that is represented by a migration candidate placed in the CIMDB is referred to as a *DB-op*. In addition, $ed'_{ij} \in \mathcal{ED}'$ represents the data flow from the operator v_i^x to the operator v_j^y, $x, y \in \{dsps, db\}$. Finally, $\mathcal{M}$ defines a mapping from physical query operators $\mathcal{V}'$ to logical query operators $\mathcal{V}$. For each physical query operator $v^x \in \mathcal{V}'$, $\mathcal{M}(v^x)$ defines the subset of logical query operators in $\mathcal{V}$ that the physical query operator v^x maps to. Specifically, $\mathcal{M}(v^{dsps})$ is a set which contains only one logical query operator; and $\mathcal{M}(v^{db})$ is a set which contains one or more logical query operators.

6.3 Query Optimization

This section describes the static cost-based query optimizer proposed in this dissertation to determine the query execution plan for a given continuous query for hybrid query execution as shown in Figure 6.2. Specifically, Section 6.3.1 defines the query optimization objective in such hybrid systems. Section 6.3.2 drills down to the cost model adopted by the proposed query optimizer. Section 6.3.3 describes the two-phase optimization approach applied by the optimizer, and Section 6.3.4 presents the pruning strategy applied in Phase-Two of the optimization approach.

6.3.1 The Optimization Objective

In addition to the end-to-end latency, a common performance metric for a continuous query executed over data streams is the output rate of the query [GÖ03a]. Therefore, maximizing the query output rate is a widely adopted objective in continuous-query optimization [AN04; VN02]. Maximizing the output rate of a query is equivalent to maximizing the rate of consuming the input data of the query, i.e., the *throughput* of the query. Intuitively, a query execution plan reaches its maximum throughput when it keeps up with the incoming tuple arrival rate. This capability of keeping up with the tuple arrival rate is defined as the *feasibility* of the execution plan [AN04]. A continuous query is called a *feasible query* if it has at least one feasible execution plan.

The optimization objective on the query throughput suggests that a query optimizer should favor feasible execution plans over infeasible execution plans for feasible queries, and should pick the execution plan that can maximize the query throughput for infeasible queries. However, what if a query has multiple feasible execution plans? Ayad and Naughton [AN04] have shown that given enough computation resources, all feasible execution plans of a continuous query have the same throughput. To further determine the optimality of all the feasible execution plans of a query, a different optimization objective is applied—that is, minimizing the total resource utilization of the query. The motivation behind this optimization objective is the following: intuitively, the fewer resources each query consumes, the more number of queries that a system can execute concurrently. In summary, the optimization objective of the optimizer in the proposed hybrid system is defined as follows:

- For a feasible continuous query, find the feasible execution plan that has the least resource utilization.
- For an infeasible continuous query, find the execution plan that has the maximum query throughput.

Generally speaking, given two execution plans of a continuous query, the possible situations that the query optimizer may face, as well as the respective, appropriate optimization decisions, are the following:

- Situation 1: One execution plan is feasible and the other is infeasible. → Choose the feasible execution plan.
- Situation 2: Both execution plans are feasible. → Choose the plan that has a lower resource utilization.

- Situation 3: Both execution plans are infeasible. → Choose the plan that has a higher query throughput.

Discussion Ayad and Naughton [AN04] adjusted the above optimization objective to incorporate the influence of load shedding (cf. Section 2.3.2 and Section 3.5). In their work, load shedding operators were inserted into execution plans of an infeasible query, thereby transforming all infeasible execution plans into feasible plans. The work presented in this chapter focuses on discussing the optimization of continuous queries in a hybrid system and does not consider applying load shedding for infeasible queries.

6.3.2 The Cost Model

To achieve the optimization objective defined in Section 6.3.1, a cost-based optimizer is proposed. Without loss of generality, let us consider continuous queries whose logical plans have window operators immediately follow source operators, and have sink operators immediately follow R2S operators. A query that has window or R2S operators appear as internal nodes in the logical query plan can always be split into a set of sub-queries, with the logical plan of each sub-query satisfying the above constraint. In addition, it is assumed that the system is deployed in a highly parallel environment with abundant memory. Hence, the query operators can be fully pipelined and do not time-share CPU resources.

The tuple arrival rate r_i of an input stream S_i involved in a query execution plan $\mathcal{P}$ defines how many tuples from the stream S_i should be processed by $\mathcal{P}$ within a unit of time. In the following, the tuples arrived from all input streams involved in an execution plan within a unit of time are referred to as the *unit-time source arrivals*, denoted by λ^{in}. Furthermore, the amount of tuples that an operator produces as a result of the unit-time source arrivals is referred to as the *source-driven output-size* of the operator, denoted by λ^{out}. Note that (1) the source-driven output-size of an operator is the amount of tuples produced by the operator as a result of the unit-time source-arrivals, rather than the amount of tuples generated by the operator within a unit of time, which is also known as the output rate; (2) the source-driven output-size of an operator v_i is equal to the source-driven input-size of its direct downstream operator v_j, and is used to estimate the source-driven output-size of the operator v_j.

Given the tuple arrival rates of all input streams involved in a continuous query, the source-driven output-size λ^{out} of each operator in the query can be estimated in a cascaded way, starting from the source operators. To estimate λ^{out} of window-based selection, projection, and join, the proposed optimizer adapts the method proposed in [AN04] based on the query semantics model defined in cf. Section 2.1. Specifically, for a selection or a projection with a selectivity of f[1], the source-driven output-size is estimated as

$$\lambda^{out} = f \cdot \lambda^{in}. \tag{6.1}$$

For a join operator, let $|w_L|$ denote the cardinality of the operator's left input instantaneous window w_L (i.e., the number of tuples contained in w_L), $|w_R|$ denote the cardinality of the operator's right input instantaneous window w_R, and f_L, f_R denote the selectivities relative to w_L and w_R, respectively. The source-driven output-size of

[1]The selectivity of a projection operator is 1.

this join operator can be estimated by Eq. (6.2). The cardinality of an instantaneous window can be estimated in a cascaded way as described in [AN04].

$$\lambda^{out} = \lambda_L^{in} \cdot f_R \cdot |w_R| + \lambda_R^{in} \cdot f_L \cdot |w_L| \tag{6.2}$$

Recall that a window-based aggregate operator produces results at each slide of the window (cf. Section 2.1). For a time-based sliding window, if the slide size is β time units, then the average sliding frequency within a unit of time, denoted by n_{slide}, is $\frac{1}{\beta}$. For a count-based sliding window whose slide size is β tuples, the sliding frequency depends also on the tuple arrival rate r of the input stream, and can be estimated as $n_{slide} = \frac{r}{\beta}$. The aggregate operator may have an associated grouping predicate. Suppose that the average number of result groups is n_{group}. The source-driven output-size of an aggregate operator can then be estimated as

$$\lambda^{out} = n_{slide} \cdot n_{group} \tag{6.3}$$

Cost of an Individual Operator

Based on the above estimation of the source-driven input/output-sizes of operators, the cost of a physical operator in an execution plan can be estimated. Each tuple arriving at a physical operator requires a certain amount of processing effort from the operator. The average time that an operator v_j^x requires to process a single tuple from a direct upstream operator v_i^x is referred to as the *unit processing cost of v_j^x for v_i^x*, denoted by c_{ji}, or simply c_j if the operator v_j^x has only one direct upstream operator. For an operator v_j^x with l direct upstream operators, the total cost of v_j^x caused by unit-time source-arrivals is referred to as the *source-driven input processing cost*, denoted by u_j^x. In general, u_j^x is estimated as

$$u_j^x = \sum_{i=1}^{l} \lambda_i^{in} \cdot c_{ji}, \tag{6.4}$$

where l represents the number of direct upstream operators of the operator v_j^x.

To keep up with the tuple arrival rates of all input streams, the time needed to process a single tuple by each physical operator in an operator pipeline must be shorter than the average tuple arrival interval at the operator. In other words, the constraint $\sum_{i=1}^{l} \lambda_i^{in} \cdot c_{ji} \leq 1$, namely $u_j^x \leq 1$, must be satisfied [AN04; VN02]. An operator that cannot meet this constraint is referred to as a *bottleneck* of the operator pipeline.

Cost of a *DSPS-op* The cost-estimation method described above can be used directly to estimate the costs of *DSPS-op*s (cf. Section 6.2) in an execution plan. The unit processing cost c of a specific *DSPS-op* depends on the type and the physical implementation of the operator. In our prototype system, the unit processing cost c of a selection operator and a projection operator is modeled as a constant. The reason is that for these two types of operators, processing a tuple only needs to inspect the tuple itself. For a join operator, the unit processing cost c is modeled as a function of the cardinality of the instantaneous windows over which the join operation is performed. Here, the model proposed in [KNV03] is adopted. For an

aggregate operator, the unit processing cost c is either a constant if the aggregate can be computed incrementally (e.g., COUNT, SUM, and AVG), or a function of the cardinality of the input instantaneous window if the aggregate cannot be evaluated incrementally (e.g. MEDIAN).

Cost of a *DB-op* In contrast to a *DSPS-op*, which maps to a single logical query operator, a *DB-op* maps to one or more logical query operators and is evaluated as a single SQL query in the CIMDB (cf. Figure 6.2). Hence, the unit processing cost c of a *DB-op* is practically the execution cost of the corresponding SQL query. Moreover, each time when a *DB-op* is executed, relevant input data needs to be transferred from the DSPS to the CIMDB, and the results of the SQL query need to be transferred back from the CIMDB to the DSPS. The costs of these inter-engine data transfers must be taken into account as well. In summary, the unit processing cost of a *DB-op* consists of three parts: the cost of transferring the relevant input data from the DSPS to the CIMDB, the cost of evaluating the corresponding SQL query in the CIMDB, and the cost of transferring the results of the SQL query back to the DSPS. In the prototype system built in this dissertation, the built-in cost model of the CIMDB was extended and tuned to estimate the cost of a *DB-op*.

Note that although the CIMDB may exploit multiple types of parallelization opportunities (e.g., pipelined parallelism and partition parallelism [Gra93]) to evaluate the SQL queries corresponding to the *DB-op*s, it is assumed that the performance impact caused by parallelization is already considered by the cost model of the CIMDB. How to build an accurate cost model for a parallel database is beyond the scope of this dissertation.

Cost of an Execution Plan

The cost of a complete query execution plan can be estimated based on the costs of individual physical query operators as estimated above. Corresponding to the optimization objectives defined in Section 6.3.1, the cost of an execution plan $\mathcal{P}$ with $|\mathcal{V}'|$ operators, denoted by $C(\mathcal{P})$, is defined as a two dimensional vector consisting of two cost metrics: the *bottleneck cost* $C_b(\mathcal{P})$ and the *total utilization cost* $C_u(\mathcal{P})$; namely, $C(\mathcal{P}) = \langle C_b(\mathcal{P}), C_u(\mathcal{P})\rangle$. $C_b(\mathcal{P})$ and $C_u(\mathcal{P})$ are computed using Eq. (6.5) and Eq. (6.6), respectively.

$$C_b(\mathcal{P}) = max\{u_j^x : j \in [1, |\mathcal{V}'|]\}, x \in \{dspe, db\}. \tag{6.5}$$

$$C_u(\mathcal{P}) = \sum_{j=1}^{|\mathcal{V}'|} u_j^x, , x \in \{dspe, db\} \tag{6.6}$$

Here, "bottleneck" refers to the operator in an execution plan that has the highest source-driven input processing cost. The bottleneck cost $C_b(\mathcal{P})$ is used to check the feasibility of an execution plan. Moreover, for infeasible execution plans of a query, a higher bottleneck cost implies that the execution plan can handle fewer input tuples per unit of time. Therefore, the bottleneck cost is also used as an indicator of the throughput of an infeasible execution plan. The total utilization cost $C_u(\mathcal{P})$ estimates the total amount of resources required by an execution plan to process unit-time source arrivals.

Based on the above cost model for a query execution plan, the *optimal execution plan* of a given continuous query is defined as follows:

Definition 6.1 (Optimal Query Execution Plan)**.** *For a continuous query Q, an execution plan $\mathcal{P}$ is an optimal execution plan of Q, if and only if for any other execution plan $\mathcal{P}'$ of the query Q, one of the following conditions is satisfied:*

- *Condition 1°:* $C_b(\mathcal{P}) \leq 1 < C_b(\mathcal{P}')$
- *Condition 2°:* $C_b(\mathcal{P}) \leq 1$, $C_b(\mathcal{P}') \leq 1$, *and* $C_u(\mathcal{P}) \leq C_u(\mathcal{P}')$
- *Condition 3°:* $1 < C_b(\mathcal{P}) \leq C_b(\mathcal{P}')$

Each condition in Definition 6.1 is applied in a specific situation described in Section 6.3.1. Specifically, Condition 1° is applied when the plan $\mathcal{P}$ is feasible and the plan $\mathcal{P}'$ is infeasible; Condition 2° is applied when both $\mathcal{P}$ and $\mathcal{P}'$ are feasible; and Condition 3° is applied when both $\mathcal{P}$ and $\mathcal{P}'$ are infeasible.

6.3.3 Two-Phase Optimization

In principle, a R2R operator of a continuous query can be executed either in the DSPS or in the CIMDB in Figure 6.2. However, the placement decision for the operator does not influence the pipelined relationships between this operator and its upstream and downstream operators. Consequently, the choices of the processing engine for a logical query operator can be treated as physical implementation-alternatives of the operator [Bla+05], which allows integrating the selection of the processing engine for logical query operators into the phase of enumerating the execution plans of a query optimizer.

A continuous query could have a large number of semantically-equivalent logical plans due to, for instance, different join ordering possibilities. Even for an individual logical plan G with n R2R operators, there are in total 2^n possible execution plans for G. Because of the large search space of query execution plans, exhaustive search for the optimal execution plan is prohibitive. Following the idea applied in many existing federated, distributed, or parallel database systems, the optimizer proposed in this dissertation adopts a *two-phase* optimization approach [HS91]. Specifically, the optimization process is divided into Phase-One and Phase-Two. In Phase-One, the optimizer assumes that all operators are placed in the DSPS, and then determines the optimal execution plan for a given continuous query under this assumption, considering the join ordering and the push-down (or pull-up) of aggregates, etc. In Phase-Two, the optimizer takes the logical query plan corresponding to the plan obtained in Phase-One as input, and determines the placement of each operator in that logical plan over the two underlying processing engines.

The *System R* style dynamic-programming optimizer [Sel+79] is used widely in existing database systems. This type of query optimizer relies on the so-called *principle of optimality* to prune away expensive query execution plans as early as possible. Our optimizer adopts the *System R* style optimization approach to find the optimal execution plan in Phase-One. However, to guarantee that this adoption is applicable, it must be shown that the principle of optimality holds in the context of continuous-query optimization as well; namely, the optimal execution plan for

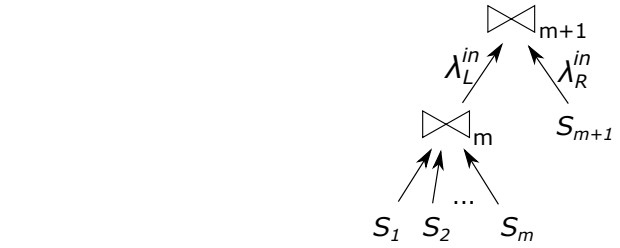


Figure 6.3: Illustrative execution plan which extends the subplan joining a set of streams $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$ to join with another stream S_{m+1}.

joining a set of m streams $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$ with another stream S_{m+1} can be obtained by joining the stream S_{m+1} with the optimal execution plan that joins all streams in $\mathbb{S}$.

Let us consider the join query in Figure 6.3. The window operators are omitted for brevity. Let $\mathcal{P}_{opt}$ denote the optimal execution plan for joining the set of streams $\mathbb{S} = \{S_1, S_2, \ldots, S_m\}$. Any suboptimal execution plan is denoted as $\mathcal{P}_s$. Suppose that the next stream to be joined is S_{m+1}, which incurs λ_R^{in} unit-time source-driven arrivals at the new join operator denoted by $\bowtie_{m+1}$. Note that the total number of join results produced by the optimal plan $\mathcal{P}_{opt}$ as a result of the unit-time arrivals from all streams in $\mathbb{S}$ is the same as that produced by any sub-optimal plan $\mathcal{P}_s$. Namely, the source-driven output-size of the join operator $\bowtie_m$ is identical in all execution plans that join streams in $\mathbb{S}$. Hence, according to Eq. (6.4), it can be inferred that the source-driven input processing cost u_{m+1} of the join operator $\bowtie_{m+1}$ is identical in all execution plans that are extended from the execution plans for $\bowtie_m$. Denoting the execution plan extended from $\mathcal{P}_{opt}$ to join with the stream S_{m+1} as $\mathcal{P}'_{opt}$, and the execution plan extended from $\mathcal{P}_s$ to join with S_{m+1} as $\mathcal{P}'_s$, in the following, it is proved that the plan $\mathcal{P}'_{opt}$ is still optimal compared to any $\mathcal{P}'_s$.

Proof Sketch.
Case 1: The plan $\mathcal{P}_{opt}$ is feasible. In this case, an execution plan $\mathcal{P}_s$ is suboptimal either because it is infeasible (i.e., Condition 1° in Definition 6.1), or because it is feasible as well but its total utilization cost is higher than that of $\mathcal{P}_{opt}$ (i.e., Condition 2° in Definition 6.1).

- *Case 1.1:* If the plan $\mathcal{P}_s$ is infeasible, then the execution plan $\mathcal{P}'_s$ extended from $\mathcal{P}_s$ with the join operator $\bowtie_{m+1}$ is still infeasible. Extending the execution plan $\mathcal{P}_{opt}$ with the operator $\bowtie_{m+1}$ can either leave the resulting execution plan $\mathcal{P}'_{opt}$ feasible if the source-driven input processing cost u_{m+1} of $\bowtie_{m+1}$ satisfies $u_{m+1} \leq 1$, or make $\mathcal{P}'_{opt}$ infeasible if $u_{m+1} > 1$. In the former case, it is obvious that the plan $\mathcal{P}'_{opt}$ is better than the plan $\mathcal{P}'_s$. In the latter case, the bottleneck costs of $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ need to be compared to determine which one is optimal. The bottleneck cost $C_b(\mathcal{P}'_{opt})$ of the plan $\mathcal{P}'_{opt}$ is u_{m+1}. The bottleneck cost $C_b(\mathcal{P}'_s)$ of the plan $\mathcal{P}'_s$ is $C_b(\mathcal{P}_s)$ if $u_{m+1} < C_b(\mathcal{P}_s)$, or u_{m+1} if $u_{m+1} \geq C_b(\mathcal{P}_s)$. In either case, the relationship $1 \leq C_b(\mathcal{P}'_{opt}) \leq C_b(\mathcal{P}'_s)$ exists. Therefore, according to Condition 3° in Definition 6.1, the plan $\mathcal{P}'_{opt}$ is still optimal.

- *Case 1.2:* If the plan $\mathcal{P}_s$ is feasible as well but its total utilization cost is higher than that of the plan $\mathcal{P}_{opt}$ (i.e., $C_u(\mathcal{P}_s) > C_u(\mathcal{P}_{opt})$), then the feasibilities of the extended plans $\mathcal{P}'_{opt}$ and $\mathcal{P}'_s$ are determined by the source-driven input processing cost u_{m+1} in the same way. Specifically, if $u_{m+1} \leq 1$, then both the plan $\mathcal{P}'_{opt}$ and the plan $\mathcal{P}'_s$ are feasible. Moreover, the total utilization cost $C_u(\mathcal{P}'_s)$ of the plan $\mathcal{P}'_s$ is higher than the total utilization cost $C_u(\mathcal{P}'_{opt})$ of the plan $\mathcal{P}'_{opt}$, because $C_u(\mathcal{P}'_s) = C_u(\mathcal{P}_s) + u_{m+1}$, $C_u(\mathcal{P}'_{opt}) = C_u(\mathcal{P}_{opt}) + u_{m+1}$, and $C_u(\mathcal{P}_s) > C_u(\mathcal{P}_{opt})$. Therefore, according to Condition 1° in Definition 6.1, the plan $\mathcal{P}'_{opt}$ is optimal compared to the plan $\mathcal{P}'_s$. If $u_{m+1} > 1$, then both the plan $\mathcal{P}'_{opt}$ and the plan $\mathcal{P}'_s$ are infeasible, and there exists $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}'_s) = u_{m+1} > 1$. Therefore, the plan $\mathcal{P}'_{opt}$ is still optimal according to Condition 3° in Definition 6.1.

Case 2: The plan $\mathcal{P}_{opt}$ is infeasible. In this case, the plan $\mathcal{P}_s$ can be suboptimal only when $\mathcal{P}_s$ is infeasible and the bottleneck costs of $\mathcal{P}_{opt}$ and $\mathcal{P}_s$ satisfy $1 < C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s)$ (i.e., Condition 3° in Definition 6.1). An execution plan extended from an infeasible execution plan is still infeasible. Therefore, both the plan $\mathcal{P}'_{opt}$ and the plan $\mathcal{P}'_s$ are infeasible. Depending on the value of the source-driven input processing cost u_{m+1}, the relationship between the bottleneck costs of both plans, $C_b(\mathcal{P}'_{opt})$ and $C_b(\mathcal{P}'_s)$, could be one of the following cases:

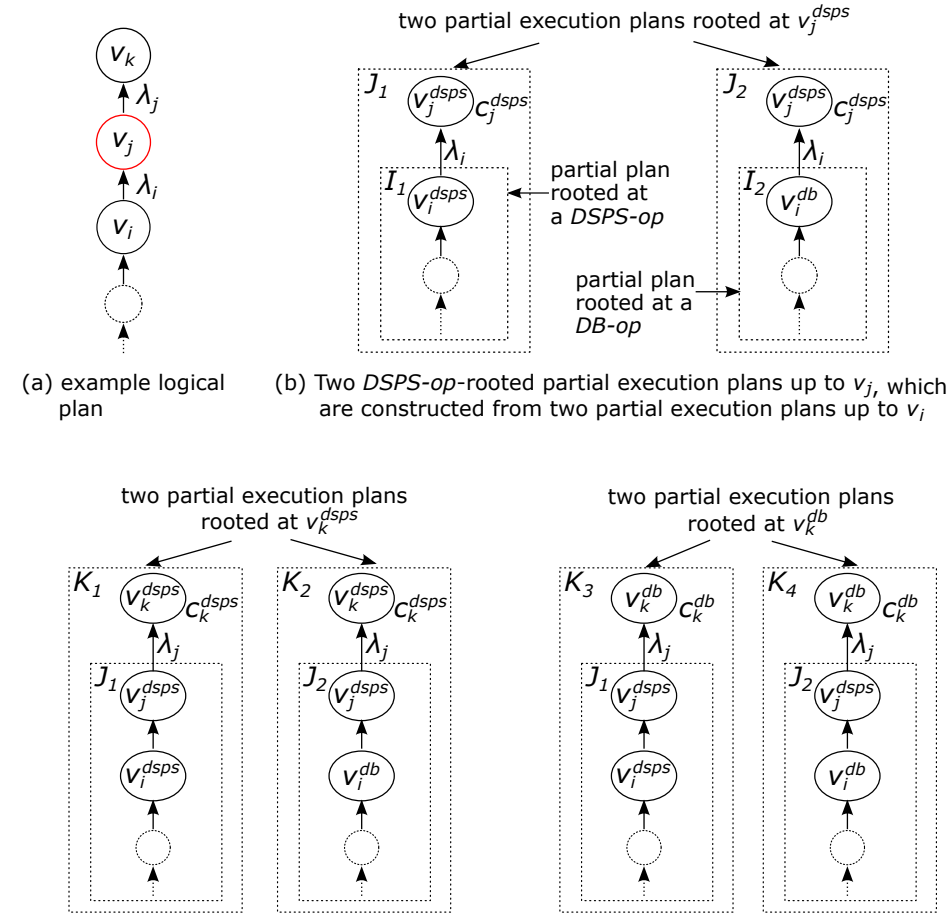
- If $u_{m+1} < C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s)$ holds, then $C_b(\mathcal{P}'_{opt})$ and $C_b(\mathcal{P}'_s)$ satisfy $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s) = C_b(\mathcal{P}'_s)$.
- If $C_b(\mathcal{P}_{opt}) \leq u_{m+1} < C_b(\mathcal{P}_s)$ holds, then $C_b(\mathcal{P}'_{opt})$ and $C_b(\mathcal{P}'_s)$ satisfy $C_b(\mathcal{P}'_{opt}) = u_{m+1} < C_b(\mathcal{P}_s) = C_b(\mathcal{P}'_s)$.
- If $C_b(\mathcal{P}_{opt}) < C_b(\mathcal{P}_s) \leq u_{m+1}$ holds, then $C_b(\mathcal{P}'_{opt})$ and $C_b(\mathcal{P}'_s)$ satisfy $C_b(\mathcal{P}'_{opt}) = C_b(\mathcal{P}'_s) = u_{m+1}$.

It can be observed that the relationship $1 < C_b(\mathcal{P}'_{opt}) \leq C_b(\mathcal{P}'_s)$ (i.e., Condition 3° in Definition 6.1) exits in all three cases. Hence, the plan $\mathcal{P}'_{opt}$ is still optimal. □

Discussion The above proof shows that the key reasons for the applicability of the principle of optimality are: (1) the source-driven input processing cost u_{m+1} of the new join operator $\bowtie_{m+1}$ is the same in all execution plans that are extended from a possible execution plan that joins the streams $S_1, S_2, \ldots, S_m$; (2) the cost u_{m+1} of the join operator $\bowtie_{m+1}$ does not change when extending $\bowtie_{m+1}$ to join with other streams.

6.3.4 Search-Space Pruning in Phase-Two of the Query Optimization

Taking the logical query plan corresponding to the execution plan produced in Phase-One of the query optimization, the proposed optimizer determines in Phase-Two the processing engine for each operator in that logical plan in a bottom-up way. Note that, different from in Chapter 5, where "bottom" refers to the sliding-window join and aggregate operators in a global logical query plan, in this chapter, "bottom" refers to the source operators in a query plan. This section describes the pruning



(a) example logical plan

(b) Two *DSPS-op*-rooted partial execution plans up to v_j, which are constructed from two partial execution plans up to v_i

(c) Two *DSPS-op*-rooted partial execution plans and two *DB-op*-rooted partial execution plans up to v_k, which are constructed based on the two partial execution plans up to v_j in (b)

Figure 6.4: Pruning opportunities when enumerating partial execution plans rooted at a *DSPS-op*.

strategy used by the optimizer in Phase-Two to further reduce the search-space size, and proves the validity of this pruning strategy.

By studying the characteristics of the cost of individual *DSPS-op*s and *DB-op*s, as well as the influence of their costs on the cost of an entire execution plan, the following properties of *DSPS-op*s are observed: (1) the source-driven input processing cost u of a *DSPS-op* v^{dsps} is identical in all partial execution plans that are rooted at the operator v^{dsps}; (2) the source-driven input processing cost of the operator v^{dsps} in a partial execution plan $\mathcal{P}$ rooted at v^{dsps} is not changed when the plan $\mathcal{P}$ is extended further with other query operators. In fact, these two properties are similar to the properties of the join operators in Figure 6.3, which suggests that a principle of optimality that is similar to the one discussed in Section 6.3.3 can be applied for pruning the search space in Phase-Two of the query optimization. Specifically, to obtain an optimal (partial) execution plan rooted at a *DSPS-op* v^{dsps}, it suffices to consider only the optimal partial execution plans rooted at the direct upstream operators of v^{dsps}.

Let us consider the logical query plan shown in Figure 6.4a. Suppose that the logical operator that is being enumerated is v_j. Because a bottom-up enumeration approach is adopted, the enumeration for the operator v_i should have completed. Suppose also that there are in total two partial execution plans up to the operator v_i, which are as shown in Figure 6.4b. Denote these two partial execution plans by I_1 and I_2. The plan I_1 is rooted at a *DSPS-op* and the plan I_2 is rooted at a *DB-op*. If plan-pruning is not applied, then two *DSPS-op*-rooted partial execution plans up to the operator v_j can be constructed; one of them extends the plan I_1, denoted by J_1, and the other one extends the plan I_2, denoted by J_2. In the following, it is proved that indeed only one *DSPS-op*-rooted partial execution plan up to the operator v_j needs to be constructed—the one that is constructed based on the optimal partial execution plan between I_1 and I_2.

Proof Sketch.
This proof consists of two parts. The first part shows that the optimality relationship between the partial execution plans J_1 and J_2 is the same as that between the partial execution plans I_1 and I_2. The second part shows that for any pair of complete execution plans $\mathcal{P}_1$ and $\mathcal{P}_2$ of the query in Figure 6.4a, the optimality relationship between $\mathcal{P}_1$ and $\mathcal{P}_2$ is the same as that between the partial execution plans I_1 and I_2, if the plans $\mathcal{P}_1$ and $\mathcal{P}_2$ differ from each other only by the partial plans up to the operator v_j, in such a way that the partial plan in the plan $\mathcal{P}_1$ is J_1 and the partial plan in the plan $\mathcal{P}_2$ is J_2.

Part 1: This part first shows that the partial execution plan J_1 is better than the partial execution plan J_2 if the plan I_1 is better than the plan I_2. According to Definition 6.1, there are three possible situations where the plan I_1 is better than the plan I_2. For each situation, the proof to show that the plan J_1 is better than the plan J_2 is similar to the proof for a specific case discussed in Section 6.3.3. Hence, here, only the references to the corresponding cases discussed in the proof in Section 6.3.3 are provided.

- Situation 1: $C_b(I_1) \leq 1 < C_b(I_2)$, i.e., the plan I_1 is feasible whereas the plan I_2 is infeasible. The proof for this situation is similar to the proof for Case 1.1 in the proof in Section 6.3.3.

- Situation 2: $C_b(I_1) \leq 1$, $C_b(I_2) \leq 1$, and $C_u(I_1) < C_u(I_2)$, i.e., both the plan I_1 and the plan I_2 are feasible, but the lower total utilization cost of the plan I_1 is lower than that of the plan I_2. The proof for this situation is similar to the proof for Case 1.2 in the proof in Section 6.3.3.

- Situation 3: $1 < C_b(I_1) \leq C_b(I_2)$, i.e., both the plan I_1 and the plan I_2 are infeasible, but the bottleneck cost of the plan I_1 is lower than that of the plan I_2. The proof for this situation is similar to the proof for Case 2 in the proof in Section 6.3.3.

The symmetric case that the plan J_2 is better than the plan J_1 if the plan I_2 is better than the plan I_1 can be proved in the same way. Moreover, the proof can be extended easily to show that for an operator v_j with multiple direct upstream operators, the

optimal *DSPS-op*-rooted partial execution plan up to v_j can be constructed from the respective optimal partial execution plans up to each direct upstream operator of v_j.

Part 2: This part shows that for a pair of complete execution plans which are constructed as extensions of the partial execution plan J_1 and the partial execution plan J_2 respectively, if the two complete execution plans differ from each other only by the partial execution plans J_1 and J_2, then the optimality relationship between them is the same as the optimality relationship between J_1 and J_2. Strictly, it needs to be shown that the optimality is retained along the execution-plan construction procedure up to the sink operator (cf. Section 2.1.2) in the logical plan. However, if it can be proved for the direct downstream operator of v_j, which is the operator v_k in Figure 6.4a, that no matter in which processing engine the operator v_k is placed, the optimality relationship between the partial execution plans extended from J_1 and J_2 is the same as the optimality relationship between J_1 and J_2, then the same reasoning can be applied recursively. Therefore, in the following, it is only shown that for the two pairs of partial execution plans, (K_1, K_2) and (K_3, K_4), in Figure 6.4c, the optimality within each pair is the same as the optimality between the plan J_1 and the plan J_2, and thus the same as the optimality between the plan I_1 and the plan I_2 as well.

For the execution-plan pair (K_1, K_2) where the operator v_k is assigned to the DSPS, the same proof in *Part 1* can be applied. The proof for the execution-plan pair (K_3, K_4) is similar. Note that in the plans K_3 and K_4, the operator v_k is placed in the CIMDB, and the source-driven input processing cost u of v_k^{db} is $\lambda_j^{out} \cdot c_k^{db}$, where c_k^{db} is the unit processing cost of v_k^{db}. If the downstream operator of v_k in the plan K_3 and the plan K_4 is placed in the CIMDB as well, then each of the two resulting partial execution plans, say K_3' and K_4', has a composite operator $v_k^{db'}$. The source-driven input processing cost u' of the composite operator $v_k^{db'}$ is $\lambda_j^{out} \cdot c_k^{db'}$, where $c_k^{db'}$ is the unit processing cost of $v_k^{db'}$. Although u' is different from u, u' is the same in both the plan K_3' and the plan K_4', and therefore does not influence the optimality relationship between K_3' and K_4'. □

Search-Space Size With the pruning strategy described above, for a logical query plan with n R2R operators, at the end of the enumeration process, only one *DSPS-op*-rooted, complete execution plan will be constructed. All the other executions plans are rooted at a *DB-op*. For a logical plan containing only unary operators, the search-space size can be reduced from 2^n to $n+1$. For a logical plan containing also binary operators, the search-space size depends heavily on the number of binary operators in the logical plan. The reason is, when constructing a *DB-op*-rooted execution plan at a binary operator v, all possibilities of combining the partial execution plans up to the left upstream operator of the operator v with the partial execution plans up to the right upstream operator of the operator v need to be considered. The worst case occurs then all n R2R operators in the logical plan are binary operators and the logical plan is a complete binary tree. Ignoring window operators, the height of the binary tree is $h = \lceil log_2(n+1) \rceil$. Given the height h of a binary tree, the upper bound of the search-space size can be defined as a function of h in a recursive way:

$$f(1) = 2$$
$$f(h) = 1 + f(h-1)^2$$

The complexity of $f(h)$ is $O(f(h)) = 2^{2^{h-1}}$. By replacing h with $\lceil log_2(n+1) \rceil$, $O(f(h))$ is approximately $2^{n/2}$, which is exponential. To be able to optimize queries with a large number of binary R2R operators with a reasonable amount of time, one solution is to decompose the logical plan produced in Phase-One into multiple subplans in such a way that each subplan has only a moderate number of binary operators. The optimizer can then optimize these subplans in their post order and construct the final execution plan by combining the optimal execution plans of the subplans.

Other Pruning Methods In addition to the pruning opportunities described above, global rules can be applied to prune candidate execution-plans during the enumeration. This approach has been applied in many existing database systems. Specifically, for instance, if after the enumeration in Phase-One, it is known that the given query is feasible, then in Phase-Two, the optimizer can stop expanding a partial execution plan as soon as the plan has been detected to be infeasible. Therefore, in Phase-One of the optimization, the optimizer can inspect the execution plan where all operators are assigned to the DSPS and the execution plan where all operators are assigned to the CIMDB. If either of them is feasible, the optimizer can perform early pruning whenever a partial execution plan is detected to be infeasible.

6.4 Evaluation

This section evaluates the proposed continuous-query optimizer from three aspects: the optimization time (Section 6.4.1), the quality of the optimization results (Section 6.4.2), and the influence of the plan-feasibility check on the quality of the optimization results (Section 6.4.3).

The DSPS and the CIMDB used in the prototype system are SAP ESP [SAP] and SAP HANA [Sik+13], respectively. The proposed optimization solution was implemented by extending the SQL optimizer of SAP HANA directly. Specifically, the cost estimation for *DSPS-op*s were added to the optimizer, and the proposed two-phase optimization approach was implemented. The prototype system was deployed on the same HP Z620 workstation as used in the evaluations of the QDDH-framework instantiations described in the previous chapters (cf. Section 4.3).

The evaluation used real-world energy-consumption data originating from smart plugs deployed in households [JZ14]. Each smart plug was uniquely identified by a combination of a *house id*, a *household id*, and a *plug id*. Each smart plug had two sensors. One sensor measured the instant power consumption, with *Watt* as the measurement unit. The other sensor measured the accumulated power consumption since the start (or reset) of the sensor, with *kWh* as the measurement unit. Each measurement was represented as a relational tuple. The type of the measurement was indicated by the *property* attribute of the tuple. Each sensor reported a measurement every 1 second, and the measurements from all smart plugs were merged into a single data stream. The original rate of this sensor data stream was approximately 2000 tuples per second. To test with higher data rates, a custom program was developed, which can replay the original sensor data at a configurable rate, to simulate different report frequencies of the sensors.

The following six continuous queries (Q_1–Q_6) were used in the evaluation:

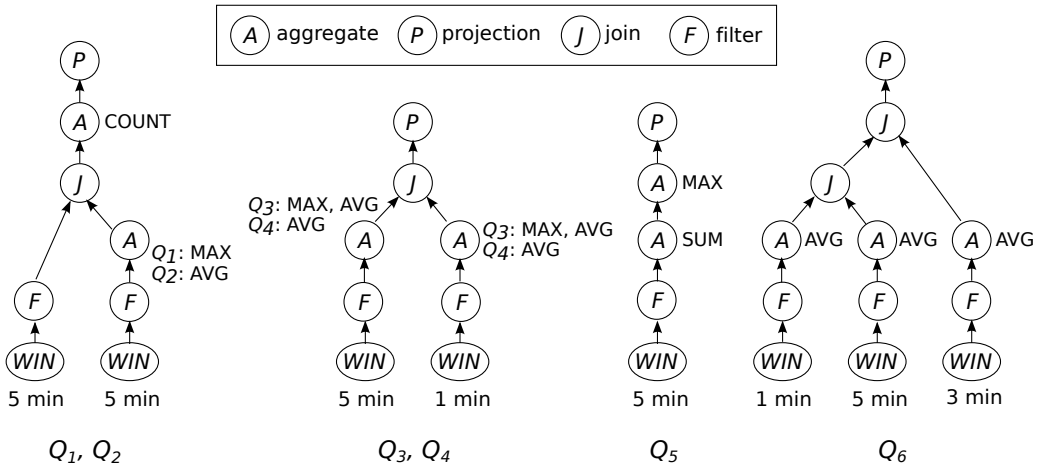


Figure 6.5: Logical plans of the queries used to evaluate the hybrid system in Figure 6.2.

- Q_1: For each smart plug, count the number of load measurements in the last 5 minutes, whose value is higher than 90% of the maximum load in the last 5 minutes.
- Q_2: For each smart plug, count the number of load measurements in the last 5 minutes, whose value is higher than the average load in the last 5 minutes.
- Q_3: For each smart plug, compare the maximum and the average loads within the last 5 minutes with the maximum and the average loads within the last 1 minute.
- Q_4: Q_4 is similar to Q_3 but compares only the average load within the two different time windows.
- Q_5: For each household, find the maximum total load reported by a single smart plug within the last 5 minutes.
- Q_6: For each smart plug, compare the average loads within the last 1, 3, and 5 minutes.

All window operators in the above queries are time-based sliding-window operators and the slide size was set to 1 second. Note that the query Q_2 and the query Q_4 were included in the query set intentionally, although they look similar to the query Q_1 and the query Q_3, respectively. The reason is that window-based AVG aggregate can be computed incrementally whereas window-based MAX aggregate cannot [Gha+07]. Hence, the cost of an AVG aggregate operator is normally lower than the cost of a MAX aggregate operator in a DSPS. The queries Q_2 and Q_4 were included to study queries with aggregate operators of different costs. Figure 6.5 shows the logical query plans devised by the optimizer described in Section 6.3 for the six queries.

	2-way join	5-way join	8-way join
Optimization time of Phase-One (ms)	0.9	68.5	100.5
# R2R op. in Phase-One produced logical plan	6	15	24
# plans examined in Phase-Two w/o pruning	64	327168	16777216
# plans examined in Phase-Two with pruning	11	312	8411
Opt. time of Phase-Two with pruning (ms)	12.3	908.6	61335.5
Total optimization time (ms)	13.2	977.1	61436

Table 6.1: Optimization times of queries with different numbers of operators.

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
Opt. time of Phase-One (ms)	1.3	1.3	0.87	0.86	22.5	5.2
Opt. time of Phase-Two with pruning (ms)	7.7	7.5	11.4	10.8	1.3	58.9
Total optimization time (ms)	9	8.8	12.27	11.66	23.8	64.1

Table 6.2: Optimization times of Q_1–Q_6 in Figure 6.5

6.4.1 Optimization Time

The first experiment examined the efficiency of the proposed optimizer in terms of the optimization time. As discussed in Section 6.3.4, the search-space size, thus the optimization time, is heavily influenced by the number of binary R2R operators in the query. Therefore, in this experiment, the query Q_4 was taken as a template, and three MSWJ queries were constructed, which compared the average loads of each smart plug within time windows of different sizes. For instance, a 5-way sliding-window join query constructed in this way first calculated the average loads of each smart plug within the last 1, 3, 5, 7, and 9 minutes, and then joined these average loads for each smart plug. In this experiment, the query-decomposition method discussed in Section 6.3.4 was not applied in the Phase-Two of the optimization. For each query, the optimization was conducted 10 times and the median of the measured optimization times was reported. The results are summarized in Table 6.1.

From Table 6.1, it can be seen that with the pruning approach described in Section 6.3.4, the number of execution plans to be examined in the Phase-Two of the optimization is reduced significantly. The results in Table 6.1 also imply that in our prototype system, it is reasonable to decompose large logical query plans into subplans with 15 operators in Phase-Two of the query optimization. With such a decomposition, the logical plan of the 8-way join query chosen by the optimizer in Phase-One can be split into two subplans, thereby reducing the optimization time from 1 minute to around 2 seconds. Table 6.1 does not show the optimization time of Phase-Two when the pruning is deactivated, because the experiment would have taken too long and is not meaningful due to the large search space. To be complete, the optimization times for the queries Q_1–Q_6 in Figure 6.5 are listed in Table 6.2.

6.4.2 Effectiveness of the Proposed Optimizer

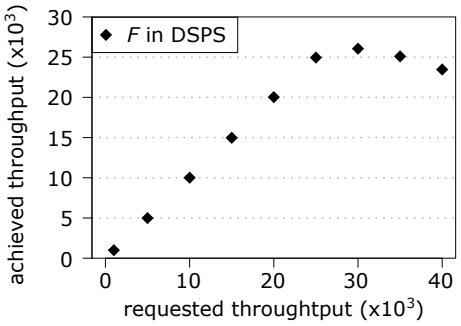
Recall from Section 6.3.2 that the proposed optimizer estimates the cost of a query execution plan based on the tuple arrival rates of the input streams involved in the

plan, and finds the optimal execution plan of a query based on the costs of execution plans. The tuple arrival rates of input streams also define the *requested throughput* of a query. To study the effectiveness of the proposed optimizer, for each query, the tuple arrival rate of the sensor data stream was varied from 1000 tuples per second to 40000 tuples per second, and the optimizer was asked to produce the optimal execution plan for each examined tuple arrival rate. Each optimal execution plan produced by the optimizer was then deployed in our prototype system, and the sensor data was pushed into the system at the corresponding rate. Then, the actual throughput of the deployed plan was measured. The results of this experiment are shown in Figure 6.6 and Figure 6.7.

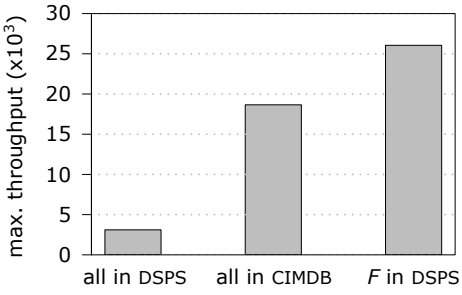
For the queries Q_1 and Q_2 (cf. Figure 6.6a–6.6d), for all examined tuple arrival rates, the optimizer picked the execution plan that placed all filters in the DSPS and the other operators in the CIMDB. The reason for this optimization decision is that both Q_1 and Q_2 computed a correlated aggregate, which required scanning the tuples within each instantaneous window twice to compute a query result. Even for a tuple arrival rate of 1000 tuples per second, a 5-minute window contains 300 thousand (in short, 300k) tuples. Frequent full scanning of instantaneous windows pushed the DSPS to its limit. In contrast, the CIMDB could compute the correlated aggregate more efficiently, despite the cost of transferring data between the two processing engines. In each of Q_1 and Q_2, the filters were placed in the DSPS to reduce the amount of data to be transferred to the CIMDB.

Let $\mathcal{P}_{dsps}$ denote a pure DSPS plan of a query, namely, all operators of the query are placed in the DSPS. Similarly, let $\mathcal{P}_{db}$ denote a pure CIMDB plan of a query, namely, all operators of the query are placed in the CIMDB. To verify the superiority of the operator-level optimization approach over the query-level optimization approach, the maximum throughputs of the optimal hybrid execution plan $\mathcal{P}_{opt}$, the pure DSPS plan $\mathcal{P}_{dsps}$, and the pure CIMDB plan $\mathcal{P}_{cimdb}$ were compared for the query Q_1 and the query Q_2. One can see from Figure 6.6b and Figure 6.6d that for both Q_1 and Q_2, the throughput of the optimal hybrid execution plan is about 8 time as high as the throughput of the pure DSPS plan. The maximum throughput of the pure CIMDB plan is also lower than that of the hybrid plan, because the pure CIMDB plan transfers more data from the DSPS to the CIMDB than the hybrid plan, which leads to a higher cost.

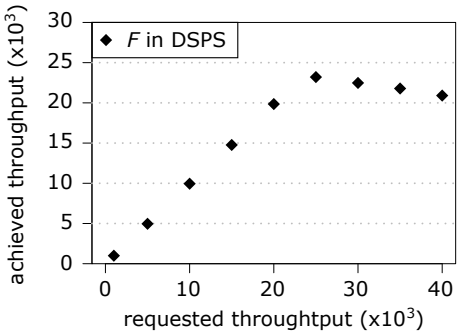
For the query Q_3 (cf. Figure 6.6e and Figure 6.6f), the execution plan that placed only filters in the DSPS (denoted by $\mathcal{P}_{opt1}$) remained optimal until the tuple arrival rate reached 20k tuples per second. For tuple arrival rates higher than 20k tuples per second, the execution plan $\mathcal{P}_{opt1}$ becomes infeasible, and the plan that placed both the filters and the join operator in the DSPS (denoted by $\mathcal{P}_{opt2}$) became optimal. Note that when the tuple arrival rate was below 20k tuples per second, the execution plan $\mathcal{P}_{opt2}$ was feasible as well. It was not picked by the optimizer because it had a higher total utilization cost compared to the execution plan $\mathcal{P}_{opt1}$. The maximum throughputs shown in Figure 6.6f confirm that the plan $\mathcal{P}_{opt1}$ became infeasible at a lower tuple arrival rate compared to the plan $\mathcal{P}_{opt2}$. When the tuple arrival rate was 20k tuples per second, the actual throughput of the plan $\mathcal{P}_{opt1}$ was indeed lower than the requested throughput. This result suggests that the plan $\mathcal{P}_{opt1}$ was already infeasible at this tuple arrival rate, and the plan $\mathcal{P}_{opt2}$ should have been picked. The throughput of the plan $\mathcal{P}_{opt2}$ at the rate of 20k tuples per second is indicated by the



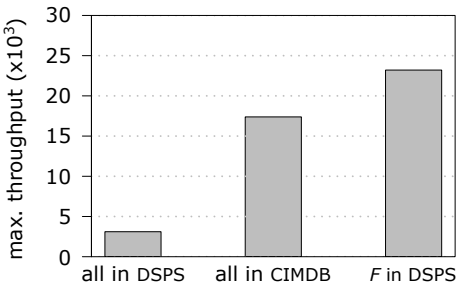
(a) Q_1: Throughput of $\mathcal{P}_{opt}$



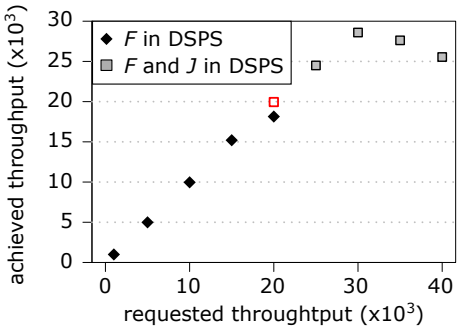
(b) Q_1: max. throughput of $\mathcal{P}_{dsps}$, $\mathcal{P}_{cimdb}$, $\mathcal{P}_{opt}$



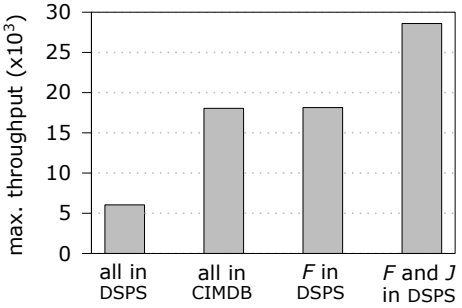
(c) Q_2: Throughput of $\mathcal{P}_{opt}$



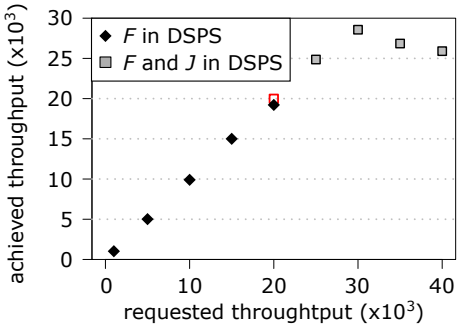
(d) Q_2: max. throughput of $\mathcal{P}_{dsps}$, $\mathcal{P}_{cimdb}$, $\mathcal{P}_{opt}$



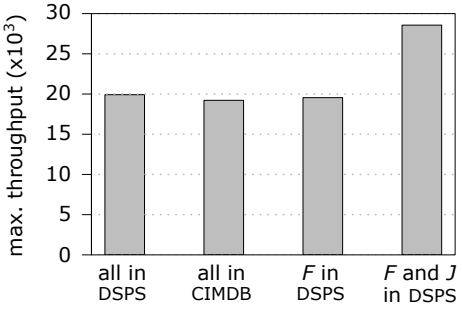
(e) Q_3: Throughput of $\mathcal{P}_{opt}$



(f) Q_3: max. throughput of $\mathcal{P}_{dsps}$, $\mathcal{P}_{cimdb}$, $\mathcal{P}_{opt}$

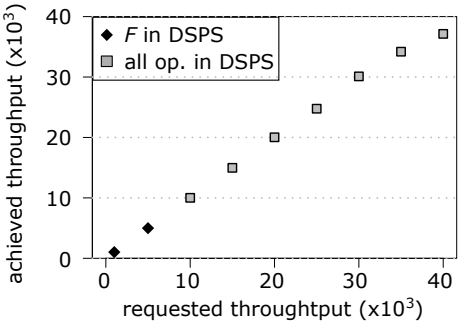


(g) Q_4: Throughput of $\mathcal{P}_{opt}$

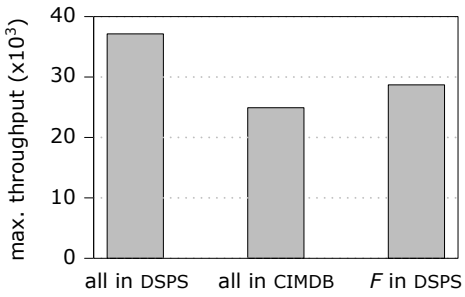


(h) Q_4: max. throughput of $\mathcal{P}_{dsps}$, $\mathcal{P}_{cimdb}$, $\mathcal{P}_{opt}$

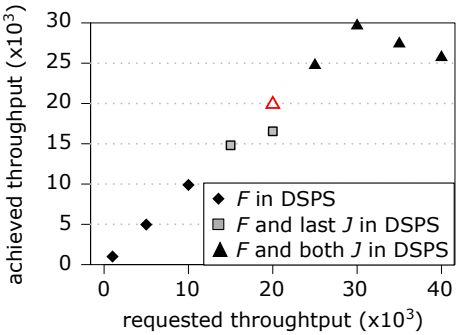
Figure 6.6: Performance of the devised optimal execution plans for the queries Q_1–Q_6 in Figure 6.5 at increasing tuple arrival rates.



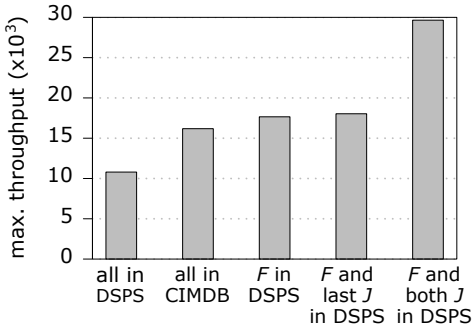
(i) Q_5: Throughtput of $\mathcal{P}_{opt}$



(j) Q_5: max. throughput of $\mathcal{P}_{dsps}$, $\mathcal{P}_{cimdb}$, $\mathcal{P}_{opt}$



(k) Q_6: Throughtput of $\mathcal{P}_{opt}$



(l) Q_6: max. throughput of $\mathcal{P}_{dsps}$, $\mathcal{P}_{cimdb}$, $\mathcal{P}_{opt}$

Figure 6.7: Performance of the devised optimal execution plans for the queries Q_1–Q_6 in Figure 6.5 at increasing tuple arrival rates. (Cont.)

hollow square in Figure 6.6e. The result that the true optimal execution plan was missed is caused by the imperfection of the cost estimation, which is an issue shared by all cost-based optimizers. However, the difference between the actual throughput of the plan $\mathcal{P}_{opt1}$ and the actual throughput of the plan $\mathcal{P}_{opt2}$ was small, and the optimizer picked the correct optimal plan successfully for every other examined tuple arrival rate. For the query Q_3, the hybrid execution plan again resulted in a higher throughput than the pure DSPS plan and the pure CIMDB plan.

The optimization results for the query Q_4 is similar to the results for the query Q_3 (cf. Figure 6.6g). However, the pure DSPS plan of Q_4 can support much higher tuple arrival rate than the pure DSPS plan of Q_3 (cf. Figure 6.6h), which confirms that computing the MAX aggregate is more expensive than computing the AVG aggregate in the DSPS.

For the query Q_5 (cf. Figure 6.7i and Figure 6.7j), the optimizer picked the execution plan that placed the filter in the DSPS when the tuple arrival rate was below $10k$ tuples per second. For higher tuple arrival rates, the total utilization cost of this execution plan became higher than the total utilization cost of the pure DSPS plan, due to the increasing cost of the data transfer between the two engines. As a result, the optimal execution plan switched to the pure DSPS plan. Unlike for the queries Q_1–Q_4, for the query Q_5, the maximum throughput of the pure DSPS plan is higher than the maximum throughputs of the hybrid plans (cf. Figure 6.7j).

The query Q_6 was a 3-way join query. Its optimal execution plan changed twice as the tuple arrival rate increased (cf. Figure 6.7k). For tuple arrival rates below $10k$

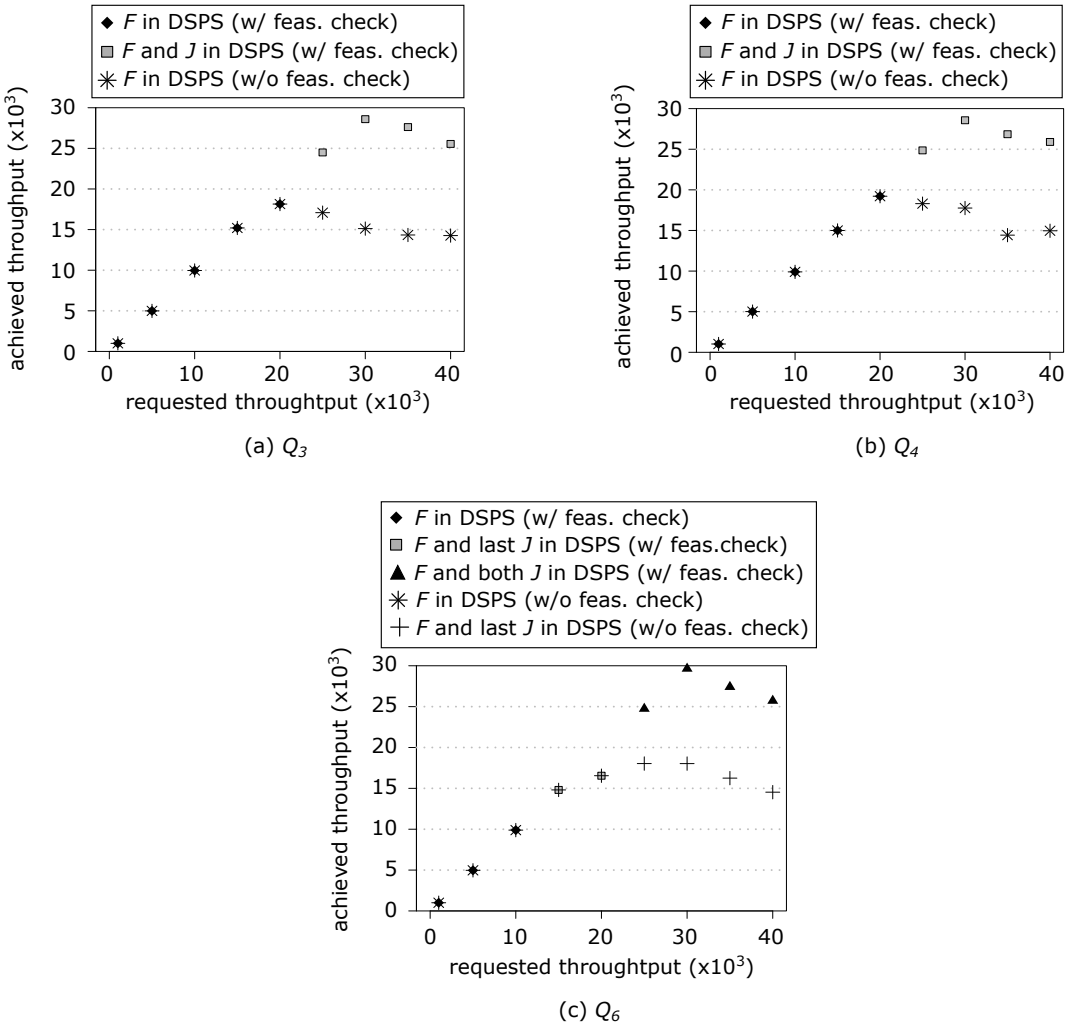


Figure 6.8: Throughput of the optimal execution plans devised with and without the plan-feasibility check.

tuples per second, the optimal execution plan placed only filters in the DSPS (denoted by $\mathcal{P}_{opt1}$). At higher tuple arrival rates up to 20k tuples per second, in addition to the filters, the optimal execution plan placed the last join operator in the query in the DSPS as well (denoted by $\mathcal{P}_{opt2}$). For even higher tuple arrival rates, only the aggregate operators were left in the CIMDB (denoted by $\mathcal{P}_{opt3}$). The switch from the plan $\mathcal{P}_{opt1}$ to the plan $\mathcal{P}_{opt2}$ was because of the higher total utilization cost of the plan $\mathcal{P}_{opt1}$ compared with the plan $\mathcal{P}_{opt2}$; and the switch from the plan $\mathcal{P}_{opt2}$ to the plan $\mathcal{P}_{opt3}$ was because of the infeasibility of the plan $\mathcal{P}_{opt2}$ (cf. Figure 6.7l). Similar to the case for the queries Q_3 and Q_4, the optimizer missed the true optimal execution plan at the tuple arrival rate of 25k tuples per second, as indicated by the hollow triangle in Figure 6.7k.

In summary, the optimizer proposed in this chapter for hybrid execution of continuous queries performs well with respect to the quality of the optimization results. Especially, for each examined query, when the tuple arrival rate of the input stream

was so high that the query becomes infeasible, the optimizer was able to pick the execution plan that can maximize the query throughput.

6.4.3 Influence of the Plan-Feasibility Check

Recall from the definition of the optimal query execution plan (i.e., Definition 6.1) that, the optimizer checks both the feasibility and the total resource utilization of possible execution plans to determine the optimal execution plan. This section studies the influence of the plan-feasibility check on the quality of the produced optimization results. To this end, the feasibility check for execution plans was turned off in the optimizer; namely, the optimization decisions were made based on only the utilization costs of execution plans. Then, the tests described in Section 6.4.2 were repeated for all six queries. For each query and each examined tuple arrival rate, the actual throughput of the optimal execution plan devised with the plan-feasibility check was compared with the actual throughput of the optimal execution plan devised without the plan-feasibility check. The experimental results are shown in Figure 6.8.

For the queries Q_1, Q_2, and Q_5, the optimization results obtained without the plan-feasibility check were identical to the optimization results obtained with the plan-feasibility check. However, for the queries Q_3 and Q_4 (cf. Figure 6.8a and 6.8b), without the plan-feasibility check, the optimizer picked the execution plan that placed only the filters in the DSPS at all examined tuple arrival rates. However, this execution plan was suboptimal compared with the plan devised with the feasibility check when the tuple arrival rate was above $20k$ tuples per second. For the query Q_6 (cf. Figure 6.8c), without the plan-feasibility check, the optimizer did not pick the execution plan that placed only the aggregate operators in the CIMDB when the tuple arrival rate was above $20k$ tuples per second, although that plan indeed had a higher throughput.

The results of this experiment confirm the necessity of the plan-feasibility check in the optimization of continuous queries. The results also imply that naive approaches for pruning partial execution-plans without considering the plan feasibility may result in suboptimal execution plans.

6.5 Related Work

Leveraging DBMSs for data stream processing has been studied in prior work including [CH10; Fra+09; LGI09]. Specifically, the Truviso system [Fra+09] integrates the continuous analytics technology into a fully functional DBMS by executing SQL queries continuously and incrementally over input data before storing the data in the database. DataCell [LGI09] is a DSPS built on top of MonetDB, an on-disk column-oriented DBMS. Chen et al. [CH10] extended the PostgreSQL DBMS to support data stream processing. This body of work focused on studying how a DBMS can be modified to support data stream processing. In contrast, the work presented in this chapter aims to leverage the advantages of both the row-oriented and the column-oriented data layout and processing techniques to improve the performance of a DSPS. The experimental results shown in Section 6.4.2 confirm the potential of such hybrid execution of continuous queries.

Note that the combination of a row-oriented DSPS and an CIMDB in our prototype system is only taken as a measure of quick system prototyping to avoid the potential

issues that may be caused by a custom implementation of the column-oriented data processing within the DSPS, although extending a DSPS directly with the capability of column-oriented data processing can eliminate the need of inter-system data transfer that our prototype system has, thereby yielding a higher system performance. Nevertheless, the query-optimization approach proposed in Section 6.3 can still be applied in such a directly-extended DSPS to determine the optimal implementation alternative of each query operator from all alternatives available in the DSPS.

MaxStream [Bot+10a] is a federated system which integrates multiple DSPSs and DBMSs. The federator layer is built on top of a relational DBMS. In MaxStream, input streams first pass through the federator layer, where the streams are persisted into, or joined with static database tables if needed. Subsequently, the streams are forwarded to a specific DSPS for query evaluation. However, MaxStream does not have an optimizer for continuous queries. ASPEN [Liu+10] is a project focusing on integrating and processing at distributed data sources including sensor devices, traditional PCs, and servers. ASPEN has a federated optimizer to assign queries across multiple subsystems. However, the optimizer does not consider the feasibility of the execution plans of a continuous query, and lacks experimental support for its effectiveness. Cyclops [LHB13] integrates Esper [Esp] (a centralized DSPS), Storm [Sto] (a distributed DSPS), and Hadoop [Apa] (a distributed batch system) for executing continuous window-based aggregate queries. Cyclops uses a black-box modeling method to build cost models. Its optimizer works at the query level, and selects the most suitable system for a given continuous query based on the window specification (i.e., the window size and the window slide) and the tuple arrival rate of the input stream. In contrast, the optimizer proposed in this chapter works at the operator level, whose superiority has been confirmed by the experimental results shown in Section 6.4.

Optimizating SQL queries in federated or distributed DBMSs [Bla+05; DH02; SL90] has been well-studied. However, existing solutions cannot be used directly to optimize continuous queries for hybrid execution as proposed in this chapter, because they do not consider the feasibility of execution plans of a continuous query. Optimization of continuous SPJ queries concerning the plan feasibility was initially studied by Viglas and Naughton [VN02]. Their work was extended by the work of [AN04], which considered the optimal placement of load-shedding operators in infeasible execution plans when the computation resources are insufficient. Cammert et al. [Cam+08] dealt with a similar resource management problem, and proposed techniques that are based on the adjustment of window sizes and time granularities. Moreover, the cost model used in [Cam+08] supports queries containing aggregation operators. However, the above work did not consider query optimization for hybrid execution with row-oriented and column-oriented processing techniques as described in this chapter.

There is a large body of work on operator placement in distributed or heterogeneous DSPSs (e.g., [LLS08; Dau+11]). This body of work normally assumes that the pipelined relationships between query operators are already determined, and considers only the placement of operators in the available processing nodes or systems. Furthermore, the feasibility-dependent optimization objective was not adopted. Nevertheless, studying how to adapt these optimization approaches in the Phase-Two optimization of the proposed optimizer is an interesting direction for future work.

6.6 Summary

Focusing on the question how to enhance a DSPS to reduce the extent of trading the query-result quality for the performance, this chapter proposed a system-enhancement approach that combines the row-oriented and the column-oriented data layout and processing techniques for evaluating continuous queries over data streams. This proposal falls into the fifth category of the system-enhance approaches discussed in Section 2.3.1—leveraging advantages of different technologies. Moreover, a static cost-based query optimization approach was introduced for such hybrid query execution. To fully exploit the potential of such hybrid execution of continuous queries, the optimizer works at the operator level and determines the optimal execution option of each operator in a query based on the characteristics of the query and the involved input streams. The proposed optimizer takes into account the feasibility of execution plans of a continuous query, as well as the non-additivity of the query-execution cost caused by the underlying heterogeneous processing techniques. The effectiveness of the optimizer was demonstrated experimentally in a prototype system composed of a row-oriented DSPS and a CIMDB. It was shown that even for simple queries, the proposed optimizer can make non-obvious decisions which resulted in up to 4 and 1.6 times higher throughput compared to the pure DSPS-based execution and the pure CIMDB-based execution, respectively. This result confirms that the proposed system-enhancement approach is viable and promising.

7
Conclusion

This chapter summarizes the main achievements of this dissertation and outlines possible directions for future work.

7.1 Summary

Due to the growing prevalence of the Internet of Things (IoT) and the growing number of applications that need to consume real-time data, data stream processing is playing an increasingly-important role in the "big data" era today. When evaluating continuous queries over data stream, a data stream processing system (DSPS) often needs to make tradeoffs between the performance of the query processing and the quality of the produced query results. The reasons are mainly twofold: (1) the streaming data being processed may be imperfect, and handling the data-imperfection within streams to obtain high-quality query results has a performance penalty; (2) the DSPS may have limited computation capacity itself, which makes it necessary to trade the query-result quality for performance when the workload imposed to the system exceeds the capacity of the system, in order to keep the system from overloading. The quality of the data streams arriving at a DSPS is beyond the control of the DSPS, and therefore, the tradeoffs between the performance and query-result quality caused by the data imperfection are inevitable. In contrast, the tradeoffs caused by system limitations are reducible by enhancing the DSPS itself.

This dissertation studied how to handle the performance versus result-quality tradeoffs caused by the above two aspects of reasons, focusing first on the tradeoff caused by the data imperfection and second on the tradeoff caused by system limitations. Particularly, focusing on the problem of stream disorder, which is a representative type of data imperfection in data streams, this dissertation proposed the concept of *quality-driven disorder handling* (QDDH), designed a generic buffer-based QDDH framework for processing continuous queries, and proposed methods for performing dynamic, quality-driven adaptation of the sizes of the buffers used for disorder handling at the query runtime (cf. Chapter 3). By minimizing the latency incurred by disorder handling while respecting the user-specified result-quality requirements, the proposed QDDH concept complements the state of the art and enables *making flexible and user-configurable tradeoff between the end-to-end latency and the query-result quality when dealing with the stream disorder*. Three instantiations of the generic QDDH

framework were described in detail. Two of them target individual queries of two widely-used query types—sliding-window aggregate queries and m-way sliding-window join queries (cf. Chapter 4). The third instantiation targets concurrent queries with shared operators such as source and filter operators (cf. Chapter 5). All three instantiations were implemented in a prototype DSPS which extends SAP ESP [SAP]. Experimental results showed that the proposed instantiations, thus the concept of QDDH, is effective. Compared to the state-of-the-art disorder handling approaches, the QDDH approach presented in this dissertation can significantly reduce the end-to-end latency, while still providing users with desired query-result quality.

To reduce the performance versus result-quality tradeoff caused by system limitations, this dissertation proposed a system-enhancement approach which *exploits the potential of combining both the row-oriented and the column-oriented data layout and processing techniques in data stream processing*, and designed a *static, cost-based query optimizer* to devise optimal query execution plans for such hybrid execution of continuous queries (cf. Chapter 6). By working at the query-operator level and taking the feasibility of query execution plans into account, the proposed optimizer can fully exploit the potential of such hybrid query execution. Even for simple queries, the optimizer can make non-obvious optimization decisions which lead to a throughput that cannot be matched by either type of data processing technique alone.

7.2 Outlook

The concept of QDDH proposed in this dissertation provides a new opportunity towards flexible handling of the performance versus result-quality tradeoff caused by stream disorder. Although this dissertation studied the application of the QDDH concept in three representative scenarios and showed the effectiveness of QDDH in these scenarios, there are still scenarios worth further investigation. One interesting scenario is to apply QDDH for complex individual queries that contain, for example, both aggregate and join operators, or chained aggregate operators. One major challenge in this scenario is to formulate the propagation of the result quality of the query operators along an operator pipeline. Another interesting scenario for future work is, as mentioned in Chapter 5, applying QDDH for concurrent queries with shared window-based operators. Sharing window-based operators can further eliminate the redundant work that a DSPS would otherwise have to perform concurrently, and thereby improving the system performance. However, in the meanwhile, the operator sharing may conflict with the objective of QDDH, making it unable to minimize the latency penalty caused by disorder handling for each query. Hence, a reasonable tradeoff between the performance gain obtained by operator sharing and the performance loss caused by the violation of the objective of QDDH needs to be found.

There are several possible directions to extend the work that this dissertation did in the area of enhancing a DSPS through combining the row-oriented and the column-oriented data processing techniques as well. One path is to relax the assumption of the static environment and consider runtime query re-optimization in response to changing data characteristics within the input streams. Existing work on adaptive query processing [Bab05] may be adapted here. Another path is to support multi-query optimization (MQO). Apart from finding the common fragments

across multiple queries as is done by conventional MQO methods, the sharing of the data-transfer channels between the different engines that apply different processing techniques is also an important aspect that needs to be considered. Improving the efficiency of the query optimizer is an interesting path as well. In addition, as the advance of the hardware and software technologies, there is nearly unlimited potential for enhancing a DSPS to push further the boundary of the system with respect to the bearable workload.

Symbols

Notation	Definition (Section)	Description
$\mathbb{T}$	2.1.1	Time domain
S_i	2.1.1, 5.1	The i-th input stream or the source operator corresponding to the i-th input stream; $i \in [1, m]$
r_i		The tuple arrival rate of the stream S_i
$e_{i,j}$	2.1.1	The j-th arrived tuple in a stream S_i. The subscript j, or both subscripts, are skipped when they are not important for the discussion
$e_{i,j}.ts$	2.1.1	The timestamp of the tuple $e_{i,j}$, $e_{i,j}.ts \in \mathbb{T}$. The subscript j, or both subscripts, are skipped when they are not important for the discussion
W_i	2.1.2	The size of a sliding window applied over a stream S_i
β_i	2.1.2	The slide of a sliding window applied over a stream S_i
$w_{i,j}$	2.1.2	The j-th instantaneous window constructed over a stream S_i. The subscript j, or both subscripts, are skipped when they are not important for the discussion
$EP_l(w_{i,j})$	2.1.2	The lower endpoint of an instantaneous window $w_{i,j}$
$EP_u(w_{i,j})$	2.1.2	The upper endpoint of an instantaneous window $w_{i,j}$
$G = (\mathcal{V}, \mathcal{ED})$	2.1.2	A logical query plan of a continuous query
$v_i \in \mathcal{V}$	2.1.2	A logical query operator in G
$ed_{ij} \in \mathcal{ED}$	2.1.2	The data flow from operator v_i to operator v_j
${}^iT$	2.2.1	The local current time of a stream S_i; ${}^iT = max\{e_{i,j}.ts \mid e_{i,j} \in S_i\}$

continued on the next page

Notation	Definition (Section)	Description
$skew(S_i, S_j)$	2.2.1	The time skew between two streams S_i and S_j; $skew(S_i, S_j) = \lvert {}^iT - {}^jT \rvert$
$delay(e_{i,j})$	2.2.1	The delay of the tuple $e_{i,j}$; $delay(e_{i,j}) = {}^iT - e_{i,j}.ts$
$p^{\bowtie}$	3.1	The join condition of a join query
K_i	3.2.1	The K-slack buffer size applied over a stream S_i
T^{sync}	3.2.2	The maximum timestamp among the tuples that have been released from a synchronization buffer
K_i^{sync}	3.2.2	The buffer size within a synchronization buffer that implicitly contributes to handling the intra-stream disorder within a stream S_i
S_i'	3.3	The corresponding, disorder-handled, derived stream of an input stream S_i
$\lvert w \rvert$	3.4.1	The number of tuples included in an instantaneous window w during the query processing
$\lvert w \rvert_{true}$	3.4.1	The true number of tuples belonging to an instantaneous window w if there is no stream disorder
$Cvrg(w)$	3.4.1	The coverage of an instantaneous window w, $Cvrg(w) = \frac{\lvert w \rvert}{\lvert w \rvert_{true}}$; $Cvrg(w) \in [0, 1]$
R_i^{stat}	3.4.1	The adaptive window applied over a stream S_i for monitoring the tuple-delay statistics in S_i
D_i, D_i^K	3.4.1	The random variables representing the coarse-grained tuple delay in an input stream S_i and in the corresponding, disorder-handled, derived stream, respectively
f_{D_i}, $f_{D_i^K}$	3.4.1	The probability density functions of D_i and D_i^K, respectively
b	3.4.1	The size of a basic window used in the analytical-model-based buffer-size adaptation method (system parameter)
g	3.4.1	The K-search granularity used in the analytical-model-based buffer-size adaptation method (system parameter)
$w_{i,j}^l$, $\lvert w_{i,j}^l \rvert$	3.4.1	The l-th ($l \in [1, \lceil W_i/b \rceil]$) basic window in the instantaneous window $w_{i,j}$, and the cardinality of $w_{i,j}^l$, respectively
$w_{i,\top}$	3.4.1	The most recent instantaneous window constructed over a stream S_i'

continued on the next page

Notation	Definition (Section)	Description
U_p	3.4.2	The weight of the proportional term in a PD controller (system parameter)
U_d	3.4.2	The weight of the derivative term in a PD controller (system parameter)
$MaxD_i$	3.4.2	The maximum tuple delay in a stream S_i, i.e., $MaxD_i = max\{delay(e_{i,j}) \mid e_{i,j} \in S_i\}$
α	3.4.2	The parameter applied on top of $MaxD_i$ to determine the K-slack buffer size when using the PD-controller-based buffer-size adaptation method. The actual buffer size is $K_i = \alpha \cdot MaxD_i$, $\alpha \in [0, 1]$
$\Delta\alpha$	3.4.2	The adjustment, i.e., the increase or the decrease, of α
$\hat{A}$	4.1.1	A produced aggregate result
A	4.1.1	The corresponding exact result of $\hat{A}$
ϵ	4.1.1	The relative error of a produced aggregate result; $\epsilon = \frac{\lvert A-\hat{A}\rvert}{\lvert A\rvert}$
(ϵ_{thr}, δ)	4.1.1	The result relative-error threshold: user-specified result-quality requirement for SWA queries. The meaning is that $Prob(\epsilon \geq \epsilon_{thr}) \leq \delta$. δ, $\delta \in (0, 1)$, is also referred to as the confidence level
L_{warmup}	4.1.2	The length of the warm-up period before performing the first buffer-size adaptation
$Cvrg_{thr}$	4.1.2	The threshold on the coverages of instantaneous windows; derived from a user-specified (ϵ_{thr}, δ); $Cvrg_{thr} \in [0, 1]$
q	4.1.4	The retrospect parameter applied in the instantiation of the QDDH framework for SWA queries; $q \in (0, 1)$ (system parameter)
$Cvrg(w, K)$	4.1.5	The estimated coverage of an instantaneous window w under a given value of K-slack buffer size K
κ	4.1.5	The optimal QDDH buffer size for a query, i.e., the minimum K-slack buffer size needed to meet the user-specified result-quality requirement associated with the query
$MaxD_i^R$	4.1.5	The maximum tuple delay observed within the window R_i^{stat}
P_{meas}	4.2.1	The user-specified result-quality measurement period
$\gamma(P_{meas})$	4.2.1	The recall of the result tuples of an MSWJ query measured based on P_{meas}

continued on the next page

Notation	Definition (Section)	Description
Γ	4.2.1	The user-specified requirement on $\gamma(P_{meas})$
Γ'	4.2.2	The instant recall requirement applied in a single buffer-size adaptation step; derived from the user-specified Γ
${}^{\bowtie}T$	4.2.2	The current maximum timestamp among the tuples received by a join operator
L_{adt}	4.2.4	The adaptation interval applied to adapt the sizes of the K-slack buffers used for an MSWJ query (system parameter)
$\gamma(L_{adt}, K)$	4.2.4	The estimated recall of join results produced within L_{adt} under a given value of the buffer size K
$N^{\bowtie}_{prod}(L_{adt}, K)$	4.2.4	The join result size within L_{adt} under a given value of the buffer size K
$N^{\bowtie}_{true}(L_{adt})$	4.2.4	The true join result size within L_{adt} if the input streams have no disorder
$N^{\times}_{prod}(L_{adt}, K)$	4.2.4	The cross-join result size corresponding to $N^{\bowtie}_{prod}(L_{adt}, K)$
$N^{\times}_{true}(L_{adt})$	4.2.4	The true cross-join result size corresponding to $N^{\bowtie}_{true}(L_{adt})$
$sel^{\bowtie}$	4.2.4	The selectivity of a join condition
$sel^{\bowtie}(K)$	4.2.4	The selectivity of a join condition under a given value of the buffer size K
$\Phi(\epsilon_{thr})$, $\Phi(\Gamma)$	4.3.1	The requirement fulfillment ratio of the results of an individual SWA or MSWJ query; used to measure the overall query-result quality produced by a disorder handling approach.
AM-adt	4.3.3	Shorthand for "analytical-model-based buffer-size adaptation method"
PD-adt	4.3.3	Shorthand for "PD-controller-based buffer-size adaptation method"
G^{glob}	5.1	A global logical query plan that contains n individual, concurrent queries
G^i	5.1	The subplan in a global query plan that is rooted at the source operator S_i
Q_j	5.1	A SWA or an *MSWJ* query covered in G^{glob}; $j \in [1, n]$
B	5.1	An individual K-slack buffer

continued on the next page

Notation	Definition (Section)	Description
$K(B)$, $\mathcal{O}_{tgt}(B)$	5.1	The size and output targets, respectively, of a K-slack buffer B
C^{glob}	5.1	A global K-slack configuration for a global query plan G^{glob}
C^i	5.1	A K-slack configuration for a subplan G^i
$mem(C^{glob})$, $mem(C^i)$	5.1	Memory costs of C^{glob} and C^i, respectively
$\mathcal{Q}(v)$	5.3	The set of queries that share an operator v
$C^{(v)}$	5.4	The optimal local K-slack configuration for handling the disorder within the output stream of a branch operator v
v_c	5.4.1	An immediate child of v
$N_F(v)$	5.4.1	The number of filter children of an operator v
f	5.4.1	The selectivity of a filter
$\kappa^s(v)$	5.4.2	The largest sharable K-slack buffer size among the queries in $\mathcal{Q}(v)$; $\kappa^s(v) = min\{\kappa_j \vert Q_j \in \mathcal{Q}(v)\}$
f_{cs}	5.4.2	The coalesced selectivity of a filter
$\mathcal{P}(G) = (\mathcal{V}', \mathcal{ED}', \mathcal{M})$	6.2	A query execution plan of a logical query graph G
$v'dsps \in \mathcal{V}'$	6.2	A basic physical operator running in a DSPS
$v'db \in \mathcal{V}'$	6.2	A migration candidate, which can be viewed as a composite R2R operator, migrated from a row-oriented DSPS to a columnar in-memory database
$ed'_{ij} \in \mathcal{ED}'$	6.2	The data flow from v_i^x to v_j^y, $x, y \in \{dsps, db\}$
$\mathcal{M}(v^x)$	6.2	The set of logical operators in G that the physical operator v^x maps to, $x \in \{dsps, db\}$
λ^{in}	6.3.2	The unit-time source arrivals of a query execution plan; defined as the number of tuples arrived from all input streams involved in the execution plan within one time unit
λ^{out}	6.3.2	The source-driven output-size of an operator in a query execution plan; defined as the amount of tuples that an operator produces as a result of the unit-time source arrivals of the execution plan

continued on the next page

Notation	Definition (Section)	Description
c_{ji}	6.3.2	The unit processing cost of a physical operator v_j^x for its upstream operator v_i^x, $x \in \{dsps, db\}$; defined as the average time that v_j^x requires to process a single tuple from v_i^x
u_j^x	6.3.2	The source-driven input processing cost of a physical operator v_j^x in a query execution plan, $x \in \{dsps, db\}$; defined as the total processing cost caused by unit-time source arrivals of the execution plan
$C_b(\mathcal{P})$	6.3.2	The bottleneck cost of a query execution plan
$C_u(\mathcal{P})$	6.3.2	The total utilization cost of a query execution plan
$C(\mathcal{P})$	6.3.2	The cost of an execution plan; $C(\mathcal{P}) = \langle C_b(\mathcal{P}), C_u(\mathcal{P})\rangle$

Index

Bibliography

[Aba+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. "Aurora: a new model and architecture for data stream management". In: *VLDB Journal* 12.2 (2003), pp. 120–139.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL continuous query language: semantic foundations and query execution". In: *The VLDB Journal* 15.2 (2006), pp. 121–142.

[Aga+13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. ACM, 2013, pp. 29–42.

[Aki+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. "MillWheel: fault-tolerant stream processing at internet scale". In: *Proc. VLDB Endow.* 6.11 (2013), pp. 1033–1044.

[Aki+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1792–1803.

[Ale+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. "The Stratosphere platform for big data analytics". In: *The VLDB Journal* 23.6 (2014), pp. 939–964.

[Alv+12] Foteini Alvanaki, Sebastian Michel, Krithi Ramamritham, and Gerhard Weikum. "See what's enBlogue: real-time emergent topic identification in social media". In: *Proceedings of the 15th International Conference on Extending Database Technology*. EDBT '12. ACM, 2012, pp. 336–347.

[AM04] Arvind Arasu and Gurmeet Singh Manku. "Approximate counts and quantiles over sliding windows". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '04. ACM, 2004, pp. 286–296.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. "Column-stores vs. row-stores: how different are they really?" In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. ACM, 2008, pp. 967–980.

[AN04] Ahmed M. Ayad and Jeffrey F. Naughton. "Static optimization of conjunctive queries with sliding windows over infinite streams". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. ACM, 2004, pp. 419–430.

[Apa] Apache Hadoop. URL: http://hadoop.apache.org/.

[Ara+02] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. "Characterizing memory requirements for queries over continuous data streams". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '02. ACM, 2002, pp. 221–232.

[AW04] Arvind Arasu and Jennifer Widom. "Resource sharing in continuous sliding-window aggregates". In: *Proceedings of the Thirtieth international conference on Very large data bases*. VLDB '04. VLDB Endowment, 2004, pp. 336–347.

[AY07] Charu C. Aggarwal and Philip S. Yu. "A Survey of Synopsis Construction in Data Streams". In: *Data Streams: Models and Algorithms*. Springer US, 2007, pp. 169–207.

[AY09] C. C. Aggarwal and P. S. Yu. "A survey of uncertain data algorithms and applications". In: *IEEE Trans. on Knowl. and Data Eng.* 21.5 (2009), pp. 609–623.

[Bab+04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. "Adaptive ordering of pipelined stream filters". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. SIGMOD '04. ACM, 2004, pp. 407–418.

[Bab05] Shivnath Babu. "Adaptive Query Processing in Data Stream Management Systems". PhD thesis. Stanford, CA, USA, 2005.

[Bar+07] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. "Consistent streaming through time: a vision for event stream processing". In: *Proceedings of the Third Biennial Conference on Innovative Data Systems Research*. CIDR '07. 2007, pp. 363–374.

[BDM04] B. Babcock, M. Datar, and R. Motwani. "Load shedding for aggregation queries over data streams". In: *Proceedings of the 20th IEEE International Conference on Data Engineering*. ICDE '04. IEEE, 2004, pp. 350–361.

[BG07] Albert Bifet and Ricard GavaldÃ. "Learning from time-changing data with adaptive windowing". In: *In SIAM International Conference on Data Mining*. SDM '07. 2007, pp. 443–448.

[Bha+14] Pramod Bhatotia, Umut A. Acar, Flavio P. Junqueira, and Rodrigo Rodrigues. "Slider: incremental sliding window analytics". In: *Proceedings of the 15th International Middleware Conference*. Middleware '14. ACM, 2014, pp. 61–72.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. "Breaking the memory wall in MonetDB". In: *Commun. ACM* 51.12 (2008), pp. 77–85.

[Bla+05] J.A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan, and M.-C. Wu. "Distributed/heterogeneous query processing in Microsoft SQL server". In: *Proceedings of the 21st IEEE International Conference on Data Engineering*. ICDE '05. IEEE, 2005, pp. 1001–1012.

[Bot+10a] Irina Botan, Younggoo Cho, Roozbeh Derakhshan, Nihal Dindar, Ankush Gupta, Laura M. Haas, Kihong Kim, Chulwon Lee, Girish Mundada, Ming-Chien Shan, Nesime Tatbul, Ying Yan, Beomjin Yun, and Jin Zhang. "A demonstration of the MaxStream federated stream processing system." In: *Proceedings of the 26th IEEE International Conference on Data Engineering*. ICDE '10. IEEE, 2010, pp. 1093–1096.

[Bot+10b] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. "SECRET: a model for analysis of the execution semantics of stream processing systems". In: *Proc. VLDB Endow.* 3.1-2 (2010), pp. 232–243.

[Bri+08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. "Speculative out-of-order event processing with software transaction memory". In: *Proceedings of the Second International Conference on Distributed Event-based Systems*. DEBS '08. ACM, 2008, pp. 265–275.

[BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. "Exploiting K-constraints to Reduce Memory Overhead in Continuous Queries over Data Streams". In: *ACM Trans. Database Syst.* 29.3 (2004), pp. 545–580.

[BW01] Shivnath Babu and Jennifer Widom. "Continuous queries over data streams". In: *SIGMOD Rec.* 30.3 (2001), pp. 109–120.

[Cam+08] Michael Cammert, Jurgen Kramer, Bernhard Seeger, and Sonny Vaupel. "A cost-based approach to adaptive resource management in data stream systems". In: *IEEE Trans. on Knowl. and Data Eng.* 20.2 (2008), pp. 230–245.

[CDN11] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. "An overview of business intelligence technology". In: *ACM Commun.* 54.8 (2011), pp. 88–98.

[CG07] Graham Cormode and Minos Garofalakis. "Sketching probabilistic data streams". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. ACM, 2007, pp. 281–292.

[CGM10] Badrish Chandramouli, Jonathan Goldstein, and David Maier. "High-performance dynamic pattern matching over disordered streams". In: *Proc. VLDB Endow.* 3.1-2 (2010), pp. 220–231.

[CH10] Qiming Chen and Meichun Hsu. "Experience in extending query engine for continuous analytics". In: *Proceedings of the 12th international conference on Data warehousing and knowledge discovery*. DaWaK'10. Springer-Verlag, 2010, pp. 190–202.

[Cha+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. "TelegraphCQ: continuous dataflow processing". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. SIGMOD '03. ACM, 2003, pp. 668–668.

[Cha+14] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. "Trill: a high-performance incremental query processor for diverse analytics". In: *Proc. VLDB Endow.* 8.4 (2014), pp. 401–412.

[Cha09] Surajit Chaudhuri. "Query optimizers: time to rethink the contract?" In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. SIGMOD '09. ACM, 2009, pp. 961–968.

[Che+00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. "NiagaraCQ: a scalable continuous query system for Internet databases". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. SIGMOD '00. ACM, 2000, pp. 379–390.

[CKT08] Graham Cormode, Flip Korn, and Srikanta Tirthapura. "Time-decaying aggregates in out-of-order streams". In: *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '08. ACM, 2008, pp. 89–98.

[Cod70] E. F. Codd. "A relational model of data for large shared data banks". In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[Cor+12] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. "Synopses for massive data: samples, histograms, wavelets, sketches". In: *Found. Trends databases* 4 (2012), pp. 1–294.

[CR13] Lei Cao and Elke A. Rundensteiner. "High performance stream query processing with correlation-aware partitioning". In: *Proc. VLDB Endow.* 7.4 (2013), pp. 265–276.

[Cra+02] Charles D. Cranor, Yuan Gao, Theodore Johnson, Vladislav Shkapenyuk, and Oliver Spatscheck. "Gigascope: a stream database for network applications". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. ACM, 2002, pp. 647–651.

[Dau+11] Michael Daum, Frank Lauterwald, Philipp Baumgärtel, Niko Pollner, and Klaus Meyer-Wegener. "Efficient and cost-aware operator placement in heterogeneous stream-processing environments". In: *Proceedings of the 5th ACM International Conference on Distributed Event-based System*. DEBS '11. ACM, 2011, pp. 393–394.

[Dem+07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. "Cayuga: a general ourpose Event Monitoring System". In: *Online Proceedings of the Third Biennial Conference on Innovative Data Systems Research*. CIDR '07. www.cidrdb.org, 2007, pp. 412–422.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113.

[DH02] Amol V. Deshpande and Joseph M. Hellerstein. "Decoupled query optimization for federated database systems". In: *Proceedings of the 18th IEEE International Conference on Data Engineering*. ICDE '02. IEEE, 2002, pp. 716–727.

[Dob+02] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. "Processing complex aggregate queries over data streams". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. SIGMOD '02. ACM, 2002, pp. 61–72.

[DS07] Nilesh Dalvi and Dan Suciu. "Efficient query evaluation on probabilistic databases". In: *The VLDB Journal* 16.4 (2007), pp. 523–544.

[Dul+11] Michael Duller, Jan S. Rellermeyer, Gustavo Alonso, and Nesime Tatbul. "Virtualizing stream processing". In: *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '11. Springer-Verlag, 2011, pp. 269–288.

[Els+14] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. "Scalable and adaptive online joins". In: *Proc. VLDB Endow.* 7.6 (2014), pp. 441–452.

[Esp] Esper. URL: `http://esper.codehaus.org/`.

[FC97] Christof Fetzer and Flaviu Cristian. "Integrating External and Internal Clock Synchronization". In: *Real-Time Syst.* 12.2 (1997), pp. 123–171. ISSN: 0922-6443.

[Fin82] Sheldon Finkelstein. "Common expression analysis in database applications". In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*. SIGMOD '82. ACM, 1982, pp. 235–245.

[Fli] Apache Flink. URL: `http://flink.apache.org/`.

[Fra+09] 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: *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*. CIDR '09. 2009.

[GAK12] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. "SharedDB: killing one thousand queries with one stone". In: *Proc. VLDB Endow.* 5.6 (2012), pp. 526–537.

[Gao+16] Zhipeng Gao, Weijing Cheng, Xuesong Qiu, and Luoming Meng. "A missing sensor data estimation algorithm based on temporal and spatial correlation". In: *Int. J. Distrib. Sen. Netw.* 2015 (2016), 178:178–178:178.

[Ged+07] B. Gedik, Kim-Lung Wu, P.S. Yu, and Ling Liu. "A load shedding framework and optimizations for m-way windowed stream joins". In: *Proceedings of the 23rd IEEE International Conference on Data Engineering*. ICDE '07. IEEE, 2007, pp. 536–545.

[Ged+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. "SPADE: the system s declarative stream processing engine". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. ACM, 2008, pp. 1123–1134.

[Ged+14] B. Gedik, S. Schneider, M. Hirzel, and Kun-Lung Wu. "Elastic scaling for data stream processing". In: *IEEE Trans. Parallel Distrib. Syst* 25.6 (2014), pp. 1447–1463.

[Gha+07] T.M. Ghanem, M.A. Hammad, M.F. Mokbel, W.G. Aref, and A.K. Elmagarmid. "Incremental evaluation of sliding-window queries over data streams". In: *IEEE Trans. on Knowl. and Data Eng.* 19.1 (2007), pp. 57–72.

[Gia+14] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. "Shared workload optimization". In: *Proc. VLDB Endow.* 7.6 (2014), pp. 429–440.

[GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. "On computing correlated aggregates over continual data streams". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. SIGMOD '01. ACM, 2001, pp. 13–24.

[GL12] Tingjian Ge and Fujun Liu. "Accuracy-aware uncertain stream databases". In: *Proceedings of the 28th IEEE International Conference on Data Engineering*. ICDE '12. IEEE, 2012, pp. 174–185.

[GÖ03a] Lukasz Golab and M. Tamer Özsu. "Issues in data stream management". In: *SIGMOD Rec.* 32.2 (2003), pp. 5–14.

[GÖ03b] Lukasz Golab and M Tamer Özsu. "Processing sliding window multi-joins in continuous queries over data streams". In: *Proceedings of the 29th international conference on Very large data bases - Volume 29*. VLDB '03. VLDB Endowment, 2003, pp. 500–511.

[Gra93] Goetz Graefe. "Query evaluation techniques for large databases". In: *ACM Comput. Surv.* 25.2 (1993), pp. 73–169.

[Gru+10a] Le Gruenwald, Md. Shiblee Sadik, Rahul Shukla, and Hanqing Yang. "DEMS: a data mining based technique to handle missing data in mobile sensor network applications". In: *Proceedings of the Seventh International Workshop on Data Management for Sensor Networks*. DMSN '10. ACM, 2010, pp. 26–32.

[Gru+10b] Le Gruenwald, Hanqing Yang, Md. Shiblee Sadik, and Rahul Shukla. "Using data mining to handle missing data in multi-hop sensor network applications". In: *Proceedings of the Ninth ACM International Workshop on Data Engineering for Wireless and Mobile Access*. MobiDE '10. ACM, 2010, pp. 9–16.

[Gui+11] Shenoda Guirguis, Mohamed A. Sharaf, Panos K. Chrysanthis, and Alexandros Labrinidis. "Optimized processing of multiple aggregate continuous queries". In: *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. CIKM '11. ACM, 2011, pp. 1515–1524.

[Gul+12] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. "StreamCloud: an elastic and scalable data streaming system". In: *IEEE Trans. Parallel Distrib. Syst.* 23.12 (2012), pp. 2351–2365.

[HAE05] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. "Optimizing in-order execution of continuous queries over streamed sensor data". In: *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*. SSDBM'2005. Lawrence Berkeley Laboratory, 2005, pp. 143–146.

[Ham+04] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, Mohamed Y. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, Robert Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. "Nile: A Query Processing Engine for Data Streams". In: *Proceedings of the 20th IEEE International Conference on Data Engineering*. ICDE '04. IEEE, 2004, pp. 851–.

[He+09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. "Relational query coprocessing on graphics processors". In: *ACM Trans. Database Syst.* 34.4 (2009), 21:1–21:39.

[Hei+14a] Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. "Cloud-based data stream processing". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. ACM, 2014, pp. 238–245.

[Hei+14b] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. "Latency-aware elastic scaling for distributed data stream processing systems". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. ACM, 2014, pp. 13–22.

[Hir+14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. "A catalog of stream processing optimizations". In: *ACM Comput. Surv.* 46 (2014), pp. 1–34.

[HM94] Waqar Hasan and Rajeev Motwani. "Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. Morgan Kaufmann Publishers Inc., 1994, pp. 36–47.

[Hon+09] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. "Rule-based multi-query optimization". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. ACM, 2009, pp. 120–131.

[HS91] Wei Hong and Michael Stonebraker. "Optimization of parallel query execution plans in XPRS". In: *Proceedings of the First International Conference on Parallel and Distributed Information Systems*. PDIS '91. IEEE, 1991, pp. 218–225.

[Hua+08] Ming Hua, Jian Pei, Wenjie Zhang, and Xuemin Lin. "Ranking queries on uncertain data: A probabilistic threshold approach". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. ACM, 2008, pp. 673–686.

[Jai+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. "Towards a streaming SQL standard". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1379–1390.

[Jay+07] T. S. Jayram, Andrew McGregor, S. Muthukrishnan, and Erik Vee. "Estimating statistical aggregates on probabilistic data streams". In: *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '07. ACM, 2007, pp. 243–252.

[Jer+12] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. "The DEBS 2012 grand challenge". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. DEBS '12. ACM, 2012, pp. 393–398.

[JGF06] Shawn R. Jeffery, Minos Garofalakis, and Michael J. Franklin. "Adaptive cleaning for RFID data streams". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. VLDB Endowment, 2006, pp. 163–174.

[Jin+10] Cheqing Jin, Ke Yi, Lei Chen, Jeffrey Xu Yu, and Xuemin Lin. "Sliding-window top-k queries on uncertain streams". In: *The VLDB Journal* 19.3 (2010), pp. 411–435.

[JZ14] Zbigniew Jerzak and Holger Ziekow. "The DEBS 2014 grand challenge". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. ACM, 2014, pp. 266–269.

[Kar+13] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. "The HELLS-join: a heterogeneous stream join for extremely large windows". In: *Proceedings of the 9th International Workshop on Data Management on New Hardware*. DaMoN '13. ACM, 2013, 2:1–2:7.

[KBG04] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. "Detecting change in data streams". In: *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*. VLDB '04. VLDB Endowment, 2004, pp. 180–191.

[KBS06] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. "Towards correcting input data errors probabilistically using integrity constraints". In: *Proceedings of the 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access*. MobiDE '06. ACM, 2006, pp. 43–50.

[KD08] Bhargav Kanagal and Amol Deshpande. "Online filtering, smoothing and probabilistic modeling of streaming data". In: *Proceedings of the 24th IEEE International Conference on Data Engineering*. ICDE '08. IEEE, 2008, pp. 1160–1169.

[KL09] A. Klein and W. Lehner. "Representing data quality in sensor data streaming environments". In: *J. Data and Information Quality* 1.2 (2009), 10:1–10:28.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. "Evaluating window joins over unbounded streams." In: *Proceedings of the 19th IEEE International Conference on Data Engineering*. ICDE '03. IEEE, 2003, pp. 341–352.

[Kri+10] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. "Continuous analytics over discontinuous streams". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. ACM, 2010, pp. 1081–1092.

[KS04] Jürgen Krämer and Bernhard Seeger. "PIPES: a public infrastructure for processing and exploring streams". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. ACM, 2004, pp. 925–926.

[KS09] Jürgen Krämer and Bernhard Seeger. "Semantics and implementation of continuous sliding window queries over data streams". In: *ACM Trans. Database Syst.* 34.1 (2009), 4:1–4:49.

[Kul+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. "Twitter heron: stream processing at scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015, pp. 239–250.

[KWF06] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. "On-the-fly sharing for streamed aggregation". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. ACM, 2006, pp. 623–634.

[Lar+07] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. "Cardinality estimation using sample views with quality assurance". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. ACM, 2007, pp. 175–186.

[LB13] Harold Lim and Shivnath Babu. "Execution and optimization of continuous queries with Cyclops". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. ACM, 2013, pp. 1069–1072.

[Lei+15] Chuan Lei, Zhongfang Zhuang, Elke A. Rundensteiner, and Mohamed Eltabakh. "Shared execution of recurring workloads in MapReduce". In: *Proc. VLDB Endow.* 8.7 (2015), pp. 714–725.

[Lev11] William S. Levine. *The control handbook, Second Edition*. CRC Press New York, 2011. ISBN: 0-8493-8570-9.

[LGI09] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. "Exploiting the power of relational databases for efficient stream processing". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. ACM, 2009, pp. 323–334.

[LHB13] Harold Lim, Yuzhang Han, and Shivnath Babu. "How to fit when no one size fits". In: *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research*. CIDR '13. 2013.

[Li+05a] 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". In: *SIGMOD Rec.* 34.1 (2005), pp. 39–44.

[Li+05b] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. "Semantics and evaluation techniques for window aggregates in data streams". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. SIGMOD '05. ACM, 2005, pp. 311–322.

[Li+07] Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. "Event stream processing with out-of-order data arrival". In: *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*. ICDCSW '07. IEEE, 2007, pp. 67–.

[Li+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. "Out-of-order Processing: a new architecture for high-performance stream systems". In: *Proc. VLDB Endow.* 1.1 (2008), pp. 274–288.

[Lin+15] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. "Scalable distributed stream join processing". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015, pp. 811–825.

[Liu+09] Mo Liu, Ming Li, D. Golovnya, E.A. Rundensteiner, and K. Claypool. "Sequence pattern query processing over out-of-order event streams". In: *Proceedings of the 25th IEEE International Conference on Data Engineering*. ICDE '09. IEEE, 2009, pp. 784–795.

[Liu+10] Mengmeng Liu, Svilen R. Mihaylov, Zhuowei Bao, Marie Jacob, Zachary G. Ives, Boon Thau Loo, and Sudipto Guha. "SmartCIS: integrating digital and physical environments." In: *SIGMOD Rec.* 39.1 (2010), pp. 48–53.

[LLS08] Geetika T. Lakshmanan, Ying Li, and Rob Strom. "Placement strategies for internet-scale data stream systems". In: *IEEE Internet Computing* 12.6 (2008), pp. 50–60.

[LPT99] Ling Liu, Calton Pu, and Wei Tang. "Continual queries for internet scale event-driven information delivery". In: *IEEE Trans. on Knowl. and Data Eng.* 11.4 (1999), pp. 610–628.

[LZ08] Yan-Nei Law and Carlo Zaniolo. “Improving the accuracy of continuous aggregates and mining queries on data streams under load shedding”. In: *Int. J. Bus. Intell. Data Min.* 3.1 (2008), pp. 99–117.

[Mad+02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. “Continuously adaptive continuous queries over streams”. In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. SIGMOD ’02. ACM, 2002, pp. 49–60.

[Mar+04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. “The flooding time synchronization protocol”. In: *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*. SenSys ’04. ACM, 2004, pp. 39–49.

[MC08] Anurag S. Maskey and Mitch Cherniack. “Replay-based approaches to revision processing in stream query engines”. In: *Proceedings of the 2nd International Workshop on Scalable Stream Processing System*. SSPS ’08. ACM, 2008, pp. 3–12.

[MP13a] C. Mutschler and M. Philippsen. “Distributed low-latency out-of-order event processing for high data rate sensor streams”. In: *Proceedings of the 27th IEEE International Symposium on Parallel Distributed Processing*. IPDPS ’13. IEEE, 2013, pp. 1133–1144.

[MP13b] Christopher Mutschler and Michael Philippsen. “Reliable speculative processing of out-of-order event streams in generic publish/subscribe middlewares”. In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS ’13. ACM, 2013, pp. 147–158.

[MZ10] B. Mozafari and C. Zaniolo. “Optimal load shedding with aggregates and mining queries”. In: *Proceedings of the 26th IEEE International Conference on Data Engineering*. ICDE ’10. IEEE, 2010, pp. 76–88.

[MZJ13] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. “The DEBS 2013 grand challenge”. In: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. DEBS ’13. ACM, 2013, pp. 289–294.

[Neu+10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. “S4: distributed stream computing platform”. In: *2010 IEEE International Conference on Data Mining Workshops*. ICDMW ’10. IEEE, 2010, pp. 170–177.

[NSJ13] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. “Flexible query processor on FPGAs”. In: *Proc. VLDB Endow.* 6.12 (2013), pp. 1310–1313.

[Nyk+10] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. “MRShare: sharing across multiple queries in MapReduce”. In: *Proc. VLDB Endow.* 3.1-2 (2010), pp. 494–505.

[PS06] Kostas Patroumpas and Timos Sellis. “Window specification over data streams”. In: *Proceedings of the 2006 international conference on Current Trends in Database Technology*. EDBT’06. Springer-Verlag, 2006, pp. 445–464.

[Qia+13] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. "TimeStream: reliable stream computation in the cloud". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. ACM, 2013, pp. 1–14.

[QMF13] D. L. Quoc, A. Martin, and C. Fetzer. "Scalable and real-time deep packet inspection". In: *Proceedings of the 6th IEEE/ACM International Conference on Utility and Cloud Computing*. UCC '13. IEEE, 2013, pp. 446–451.

[RD08] Florin Rusu and Alin Dobra. "Sketches for size of join Estimation". In: *ACM Trans. Database Syst.* 33.3 (2008), 15:1–15:46.

[Roy+00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. "Efficient and extensible algorithms for multi query optimization". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. ACM, 2000, pp. 249–260.

[RR93] G. Ramalingam and Thomas Reps. "A categorized bibliography on incremental computation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. ACM, 1993, pp. 502–510.

[RTG14] Pratanu Roy, Jens Teubner, and Rainer Gemulla. "Low-latency handshake join". In: *Proc. VLDB Endow.* 7.9 (2014), pp. 709–720.

[Ryv+06] Esther Ryvkina, Anurag Maskey, Mitch Cherniack, and Stanley B. Zdonik. "Revision processing in a stream processing engine: a high-level design". In: *Proceedings of the 22nd IEEE International Conference on Data Engineering*. ICDE '06. IEEE, 2006, p. 141.

[SAP] SAP Event Stream Processor. URL: `http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html`.

[Sar+09] Anish Das Sarma, Omar Benjelloun, Alon Halevy, Shubha Nabar, and Jennifer Widom. "Representing uncertain data: models, properties, and algorithms". In: *The VLDB Journal* 18.5 (2009), pp. 989–1019.

[Sch+09] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. L. Wu. "Elastic scaling of data parallel operators in stream processing". In: *Proceedings of the 2009 IEEE International Symposium on Parallel Distributed Processing*. IPDPS '09. IEEE, 2009, pp. 1–12.

[SCL15] Anatoli U. Shein, Panos K. Chrysanthis, and Alexandros Labrinidis. "F1: accelerating the optimization of aggregate continuous queries". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM '15. ACM, 2015, pp. 1151–1160.

[SDS10] Thomas Schmid, Prabal Dutta, and Mani B. Srivastava. "High-resolution, low-power time synchronization an oxymoron no more". In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '10. ACM, 2010, pp. 151–161.

[Sel+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. "Access path selection in a relational database management system". In: *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. SIGMOD '79. ACM, 1979, pp. 23–34.

[Sel88] Timos K. Sellis. "Multiple-query optimization". In: *ACM Trans. Database Syst.* 13.1 (1988), pp. 23–52.

[Ses+09] S. Seshadri, Vibhore Kumar, B. Cooper, and Ling Liu. "A distributed stream query optimization framework through integrated planning and deployment". In: *IEEE Trans. Parallel Distrib. Syst* 20.10 (2009), pp. 1439–1453.

[Sha11] Zheng Shao. "Real-time analytics at Facebook". In: XLDB5 (2011). URL: `http://stanford.io/1HqxPmw`.

[Sik+13] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. "SAP HANA: the evolution from a modern main-memory data platform to an enterprise application platform". In: *Proc. VLDB Endow.* 6.11 (2013), pp. 1184–1185.

[SL90] Amit P. Sheth and James A. Larson. "Federated database systems for managing distributed, heterogeneous, and autonomous databases". In: *ACM Comput. Surv.* 22.3 (1990), pp. 183–236.

[SS94] Arun Swami and K. Bernhard Schiefer. "On the estimation of join result Sizes". In: *Proceedings of the 4th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '94. Springer-Verlag New York, Inc., 1994, pp. 287–300.

[Sto] Storm. URL: `http://storm.apache.org/`.

[Sto+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. "C-store: a column-oriented DBMS". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. VLDB Endowment, 2005, pp. 553–564.

[SW04a] Utkarsh Srivastava and Jennifer Widom. "Flexible time management in data stream systems". In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '04. ACM, 2004, pp. 263–274.

[SW04b] Utkarsh Srivastava and Jennifer Widom. "Memory-limited execution of windowed stream joins". In: *Proceedings of the 30th International Conference on Very Large Data Bases - Volume 30*. VLDB '04. VLDB Endowment, 2004, pp. 324–335.

[SY93] James W. Stamos and Honesty C. Young. "A symmetric fragment and replicate algorithm for distributed joins". In: *IEEE Trans. Parallel Distrib. Syst.* 4.12 (1993), pp. 1345–1354.

[Tan+15] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. "General incremental sliding-window aggregation". In: *Proc. VLDB Endow.* 8.7 (2015), pp. 702–713.

[Tat+03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. "Load shedding in a data stream manager". In: *Proceedings of the 29th international conference on Very large data bases - Volume 29*. VLDB '03. VLDB Endowment, 2003, pp. 309–320.

[Ter+92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. "Continuous queries over append-only databases". In: *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*. SIGMOD '92. ACM, 1992, pp. 321–330.

[TM11] Jens Teubner and Rene Mueller. "How soccer players would do stream joins". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. ACM, 2011, pp. 625–636.

[TMA10] J. Teubner, R. Mueller, and G. Alonso. "FPGA acceleration for the frequent item problem". In: *Proceedings of the 26th IEEE International Conference on Data Engineering*. ICDE '10. IEEE, 2010, pp. 669–680.

[Tos+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. ACM, 2014, pp. 147–156.

[Tra+12] Thanh T. Tran, Liping Peng, Yanlei Diao, Andrew Mcgregor, and Anna Liu. "CLARO: modeling and processing uncertain data streams". In: *The VLDB Journal* 21.5 (2012), pp. 651–676.

[Tri] Trident. URL: `http://storm.apache.org/releases/1.0.0/Trident-tutorial.html`.

[Tuc+03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. "Exploiting punctuation semantics in continuous data streams". In: *IEEE Trans. on Knowl. and Data Eng.* 15.3 (2003), pp. 555–568.

[TXB06] Srikanta Tirthapura, Bojian Xu, and Costas Busch. "Sketching asynchronous streams over a sliding window". In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*. PODC '06. ACM, 2006, pp. 82–91.

[VN02] Stratis D. Viglas and Jeffrey F. Naughton. "Rate-based query optimization for streaming information sources". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. SIGMOD '02. ACM, 2002, pp. 37–48.

[VNB03] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. "Maximizing the output rate of multi-way join queries over streaming information sources". In: *Proceedings of the 29th international conference on Very large data bases - Volume 29*. VLDB '03. VLDB Endowment, 2003, pp. 285–296.

[Wan+06] Song Wang, Elke Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. "State-slice: new paradigm of multi-query optimization of window-based stream queries". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. VLDB Endowment, 2006, pp. 619–630.

[Wan+13] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. "Quantiles over data streams: an experimental study". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD. ACM, 2013, pp. 737–748.

[WR09] Song Wang and Elke Rundensteiner. "Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. ACM, 2009, pp. 299–310.

[WTZ07] Ji Wu, K.-L. Tan, and Yongluan Zhou. "Window-oblivious join: a data-driven memory management scheme for stream join". In: *Proceedings of the 19th International Conference on Scientific and Statistical Database*. SSDBM '07. IEEE, 2007, pp. 21–30.

[YP08] Yin Yang and D. Papadias. "Just-in-time processing of continuous queries". In: *Proceedings of the 24th IEEE International Conference on Data Engineering*. ICDE '08. 2008, pp. 1150–1159.

[Zah+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized streams: fault-tolerant streaming computation at scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. ACM, 2013, pp. 423–438.

[Zho+07] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. "Efficient exploitation of similar sub-expressions for query processing". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. ACM, 2007, pp. 533–544.

[ZLY08] Qin Zhang, Feifei Li, and Ke Yi. "Finding frequent items in probabilistic data". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. ACM, 2008, pp. 819–832.

[ZN42] J. G. Ziegler and N. B. Nichols. "Optimum Settings for Automatic Controllers". In: *Trans. of ASME* 64 (1942), pp. 759–768.

[ZS02] Yunyue Zhu and Dennis Shasha. "StatStream: statistical monitoring of thousands of data streams in real time". In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB '02. VLDB Endowment, 2002, pp. 358–369.