Faculty of Computer Science
Distributed System Engineering, Chair for System Engineering

Master Thesis

Quality-Driven Disorder Handling for Sliding-Window Joins

Jun Sun

Professor: Prof. Dr. Christof Fetzer
Supervisor: M.Sc. Yuanzhen Ji (SAP SE)

Submitted on June 9, 2015

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik/Institut für Systemarchitektur/Lehrstuhl Systems Engineering

Aufgabenstellung Masterarbeit

Name, Vorname:	Sun, Jun
Studiengang:	Distributed Systems Engineering
Matrikelnummer:	3933532
Thema:	***Sliding Window Joins over Out-of-order Data Streams***

Description of Work

Driven by the increasing need for processing large volumes of data that is continuously generated by financial markets, sensors embedded in environments, network infrastructures, etc., *data stream processing* has now become an important data management technology, and many stream processing engines (SPE) have been developed. In many scenarios where data streams are produced, data elements in a stream are associated with timestamps at their generation, which determine a temporal order on the data stream. However, in many applications, especially those involving distributed and network data, data may not arrive at SPEs in the strict time-order due to network delays and asynchronous communications [CKT08, KFD+10]. Such streams where data is not ordered according to timestamps are referred to as *out-of-order* streams. The correctness of results produced by many data analyzing operations relies on the order in which the incoming data is processed, therefore, SPEs must be able to deal with out-of-order data streams.

A commonly used approach for dealing with out-of-order arrivals is the so-called K*-Slack* approach [BSW04]. Essentially, it is a buffering-based approach, where a buffer of size K time units or K stream elements is used to temporarily keep and sort incoming data in timestamp-order before sending them to actual data-analysis operations. Since the first proposal of K-Slack in [BSW04], many improvements have been proposed (e.g. [MP13a, MP13b]), which try to minimize the K-Slack buffer size while still keeping a good quality of out-of-order handling. Motivated by the observation that many stream processing applications do not require 100% correct results, an on-going research in our lab is a new variant of the K-Slack approach, which adjusts the buffer size at runtime to minimize the latency introduced by the K-Slack buffer while guaranteeing that errors in the produced results do not exceed a user-specified bound. Different types of stream processing operations might use different error models for their results. For instance, for sliding window aggregations, we can use the relative error of the produced aggregate results, in relation to the actual aggregates results if all data had arrived in strict time-order. We refer to this new variant of K-Slack as *Adaptive Quality-Driven* K*-Slack*.

We have applied the adaptive quality-driven K-Slack approach for processing sliding window aggregations over out-of-order data streams. The core of this thesis is to extend our existing work to support processing sliding-window joins over out-of-order data streams as well. Sliding-window join is a fundamental operation in data stream processing [GO03]. Joining out-of-order data streams poses new challenges in terms of the trade-off between the latency of join results and the correctness of the ordering of results [Ham05]. Within this thesis, a proper model for measuring the quality of the sliding-window join results should be developed. The developed quality model should be integrated into the adaptive quality-driven K-Slack approach and applied to process sliding window joins. The solution should be implemented in a prototype SPE, which is an extension of a state-of-the-art commercial SPE. In addition, the SPE prototype currently supports only system-time based join semantics. Therefore, an extension to support application-time based join semantics is required.

Backlog

Based on the above description of the thesis scope, a following list of features should be developed:

1. Extend the existing SPE prototype which currently supports only system-time based processing semantics for sliding window joins to support application-time based processing semantics as well.
2. Develop a model for measuring the quality of the sliding-window join results and integrate the developed model with the existing adaptive quality-driven K-slack approach.
3. Integrate the overall solution for sliding window joins over out-of-order data streams into the SPE prototype. This involves making design decisions like where to hook of the disorder handling component to best fit the current architecture and implementation of the prototype.
4. The solution should be evaluated using both synthetic and real-world out-of-order data streams.

References

[BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, September 2004.

[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, pages 89–98, New York, NY, USA, 2008. ACM.

[GO03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, June 2003.

[Ham05] Moustafa A. Hammad. Optimizing in-order execution of continuous queries over streamed sensor data. In *In Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 143–146, 2005.

[KFD+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, pages 1081–1092, New York, NY, USA, 2010. ACM.

[MP13a] C. Mutschler and M. Philippsen. Distributed low-latency out-of-order event processing for high data rate sensor streams. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1133–1144, 2013.

[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, pages 147–158, New York, NY, USA, 2013. ACM.

Betreuer:	M.Sc. Yuanzhen Ji
Verantwortlicher Hochschullehrer:	Prof. Christof Fetzer
Institut:	Systemarchitektur
Lehrstuhl:	Systems Engineering
Beginn am:	1.10.2014
Einzureichen am:	15.03.2015

Verantwortlicher Hochschullehrer

Technische Universität Dresden/Fakultät Informatik

Master's Thesis Application

Private Information of the Student:
Name, First Name: Sun, Jun

born on: 8.3.1990 Matr. No.: 3 9 3 3 5 3 2

Study Course: Distributed Systems Engineering

E-Mail Address: yfiua@qq.com

Subject:

Sliding Window Joins over Out-of-order Data Streams

We agree on the above mentioned subject and we will prepare a review each:

1st Reviewer: Prof. Dr. Christof Fetzer
(always include academic title)

Professorship: Systems Engineering

☐ I supervise the thesis work

(Date, Signature)

2nd Reviewer: ~~M.Sc. Yuanzhen Ji~~ Prof. Härtig
(always include academic title)

Professorship:

☐ I supervise the thesis work

(Date, Signature)

Start date: 1.10.2014 Submission deadline: 15.3.2015

The Examination Board accepts the application:

Date: 12.09.2014 Chair of Examination Board:

Distribution after return: Original Student, Copy 1st Reviewer, Copy 2nd Reviewer, Copy SCIS

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Service Center for International Students

Application for Extension of Submission Deadline for Master's Thesis
(maximal 3 month for CE, maximal 13 weeks for DSE)

Name: Sun	First Name: Jun
Date of Birth: 08.03,1990	Semester: 5
Student ID: 3933532	Program: DSE

Advisor:	Ji, Yuanzhen , Prof. Fetzer
Start Date:	1.10.2014
End Date:	15.3.2015
Duration of Extension:	~~3 months~~ 13 Weeks
New End Date:	~~15~~.6.2015 geä. Leö H.

Explanation for Extension:
We plan to optimize the model, according to the evaluation result.

Confirmation of Advisor: Yuanzhen Ji

This application has to be submitted in time to the examination office;
a copy of the thesis topic is to be concluded.

14.1.2015
Date

S. Leöffler-Isack
Signature of SCIS Datum entspricht nicht PO !!

Decision of the Examination Board:

The application for extension is / ~~is not~~ approved.

29.01.2015
Date

Signature
of Examination Board

Abstract

In data stream processing systems, stream processing engines (SPE) can perform various types of operations, e.g., AGGREGATE, UNION, JOIN, over incoming data streams. When the arrival order of the data is the same as the order of their generation, the SPE can produce results with no extra effort. However, it is often possible that the arrival order cannot be guaranteed due to uncontrollable factors such like network delay, thus the SPE is no longer able to keep the result semantic of the origin without exploiting disorder handling approaches, e.g., a buffer to wait for late arriving data, which causes result latency.

In this paper, we concentrate on the operation of sliding-window JOIN. In order to minimize the result latency while giving a high-quality result satisfying users' demands, we use the state-of-the-art quality metric for data stream JOIN operation, and propose a probabilistic model based on the observation of late arrival patterns that the incoming data streams exhibit. We apply this model to the SPE, in order to do online trade-off between result latency and result quality, i.e., to adjust the buffer size on the fly.

We implement our proposal in SAP Sybase Event Stream Processor (ESP). Experimental evaluations with real-world out-of-order data streams validate the effectiveness of our approach, which causes only modest computational overhead for result quality monitoring and buffer size adaptation.

Contents

Chapter 1

Introduction

Nowadays, data stream processing systems are widely used in different areas like sports, finance, fraud detection and so on [14]. According to the *continuous queries* [7] the users define, stream processing engines (SPE) can perform a variety of operations, e.g., AGGREGATE, UNION, JOIN over unbounded data streams and produce result streams. Among these operations, *sliding-window* JOIN is an ordinary but important one, which is used in many stream processing applications. The following example gives a typical sliding-window JOIN query with two input streams.

```
SELECT
  MAX(S.Timestamp, T.Timestamp) as Timestamp, S.ItemID as ItemID,
  S.Name, T.Order
FROM SensorOne as S KEEP 3 SECOND, SensorTwo as T KEEP 2 SECOND
SLIDE 1 SECOND
WHERE S.ItemID = T.ItemID
```

In the query given above, two data sources generate data as *tuples* associated with timestamps. For example, `SensorOne` generates tuples as `(Timestamp, ItemID, Name)`, and `SensorTwo` generates tuples as `(Timestamp, ItemID, Order)`. The SPE then joins the two input data streams containing continuous tuples from the two data sources respectively. The `KEEP` clauses define the sliding window sizes over two input streams. `SLIDE 1 SECOND` means that both windows slide every 1 second. The `WHERE` clause defines the join condition. The timestamp of a result tuple is defined as the maximum of the two timestamps of the two component input tuples[1].

Executing this query over the data streams S and T shown in Figure 1.1 we will get an output stream R. At global time 2, tuple (1, 0199, Alice) from Stream S joins tuple (2, 0199, Burger) from Stream T; and tuple (2, 0200, Bob) from

[1]There are different strategies regarding the timestamps of the result tuples. In this thesis we use the greater timestamp of two element tuples as the timestamp of the result tuple, as in [5].

Stream S joins tuple (1, 0200, Coke) from Stream T. At global time 4, tuple (3, 0201, Carol) from Stream S joins with tuple (4, 0201, Burger) from Stream T. Note that tuple (5, 0200, Dave) from Stream S does not join with tuple (1, 0200, Coke) from Stream T, because they are not in window at the same time.

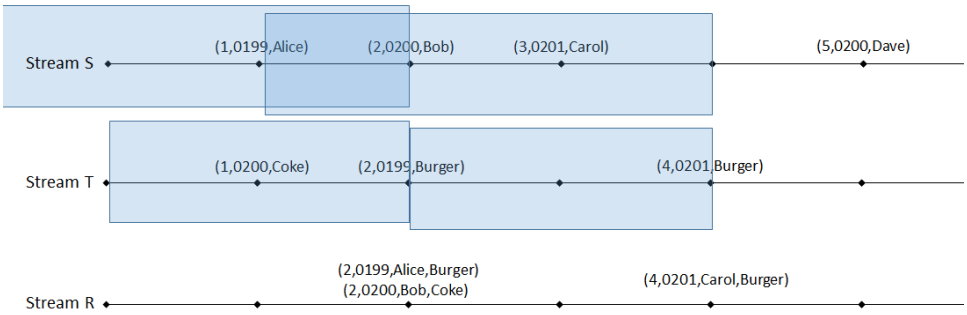


Figure 1.1: Sliding-window JOIN over two in-order data streams

In many scenarios, the temporal order in which tuples arrive are not the same as the order in which they are generated at data sources, due to unpredictable network behavior. An example is shown in Figure 1.2. This is the case especially when the SPE and the data sources are in a distributed environment. Tuple (1, 0200, Coke) from Stream T' will be discarded when it arrives at the SPE, because it is already late for the window (3, 5]. This leads to the loss of the result tuple (2, 0200, Bob, Coke).

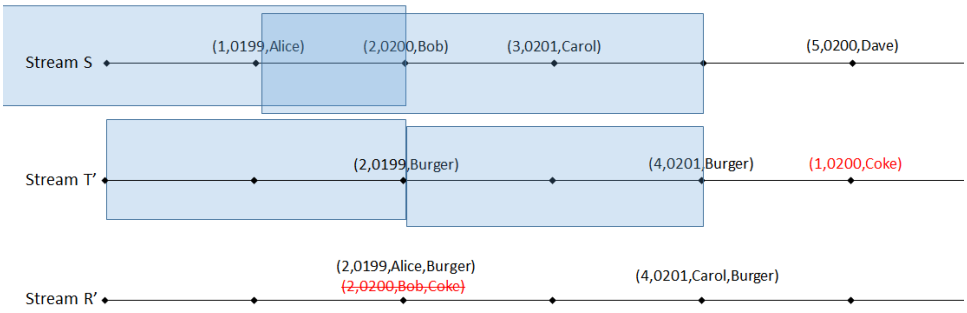


Figure 1.2: Sliding-window JOIN over out-of-order data streams

In order to keep the original semantics of the query, different disorder handling approaches have been proposed [6][8][19]. They can increase result latency to improve the result quality[2].

[2]See details in Chapter 3.

Motivated by the fact that many applications can tolerate small incorrectness of the result while having high requirement on the result latency [2], *quality-driven* disorder handling approaches are proposed, which allow user to specify an expected quality of the result, to speed up the stream processing [22]. In this thesis, we follow the quality-driven disorder handling path and focus on the *sliding-window* JOIN operation.

The contributions of this work are as follows. Using the state-of-the-art quality metric for data stream JOIN operation, we propose a probabilistic model to reveal the relationship between buffer size and result quality, when the late arrival pattern of the incoming stream data are observed. We implement the buffer size adaptation logic utilizing this model to minimize the result latency while fulfilling users' requirement on result quality. We have experimentally validated the effectiveness of our adaptation method with real-world stream data and compared it with other adaptation methods.

The rest of the thesis is organized as follows. We first introduce the preliminaries in Chapter 2 and discuss the related work in Chapter 3. Secondly, we formally describe our system model in Chapter 4. In Chapter 5, we introduce our probabilistic-based model and concentrate to describe our quality-driven adaptation algorithm. We present the implementation of our proposal and the experimental evaluation in Chapter 6. Finally, we conclude in Chapter 7.

Chapter 2

Preliminaries

In this chapter, we describe the preliminaries. We explain our time model, stream model and window model respectively.

2.1 Time Model

In this thesis, we assume that the local clocks at all data sources and at the data stream processor are always accurate and synchronized[1].

Definition 1 (Application Timestamp) The timestamp which is assigned to each tuple according to the local clock at the data source when the tuple is generated, is called an application timestamp.

Definition 2 (System Timestamp) The timestamp which is assigned to each tuple according to the local clock at the data stream processor when the tuple arrives, is called a system timestamp.

Since there is only one clock at the data stream processor, system timestamps are always monotonically increasing. Therefore discussing disorder handling based on system time is meaningless. A tuple's application timestamp is not necessarily the same as, and is usually smaller than, its system timestamp, due to many reasons like network delay. This introduces the disorder problem. We will use application timestamps in the remainder of this work unless stated specifically.

2.2 Stream Model

In this thesis, we adopt a similar stream model as defined in [9]:

[1]Nowadays, we can achieve this using GPS clocks, the Network Time Protocol, etc.

Definition 3 (Stream) A data stream is an (unbounded) sequence of tuples $e = (e.v, e.ts)$, where $e.v$ is the value of the tuple and $e.ts$ is the timestamp of the tuple.

Definition 4 (Out-of-order Stream) An out-of-order stream is a stream which contains at least one out-of-order tuple. An out-of-order tuple is a tuple that arrives later (has a greater system timestamp) than any tuple with a bigger application timestamp. Any other tuples are in-order tuples. Formally, Stream S is an out-of-order stream, iff

$$\exists e_i, e_j \in S, \text{ where } e_i.ts > e_j.ts \ \wedge \ e_i.ts_{sys} < e_j.ts_{sys}$$

The *in-order substream* for an out-of-order stream S is a stream which excludes all out-of-order tuples in S.

Definition 5 (Stream Group) A stream group is a group of at least two streams. It can be merged into a *merged* stream with the ascending order of system timestamps.

Definition 6 (Out-of-sync Stream Group) A stream group is out-of-sync, if the merged stream of the in-order substreams of all its streams is an out-of-order stream.

Definition 7 (Out-of-order Stream Group) An out-of-order stream group is a stream group whose merged stream is an out-of-order stream. It can be due to any of out-of-order stream in the group, or out-of-sync between streams.

2.3 Window Model

In order to limit the scope of interest over a potentially infinite stream, the concept of *window* has been proposed [1]. A window is a consecutive subset of a stream. Operations are applied upon the tuples within a window instead of within the whole stream. A window can have a size W, which can be either time-based (e.g., 5 seconds), or tuple-based (e.g., 10 tuples). We use the same definitions of time-based window and defined in [5].

Definition 8 (Time-based Window) At any time instant t, a time-based window of size W on a stream S defines a subset of S containing all elements of S with timestamp ts such that $t - ts \leq W$.

Definition 9 (Tuple-based Window) At any time instant t, a tuple-based window of size W on a stream S defines a subset of S with the largest W timestamps not exceeding t. If the size of S at time t is less than W, the window includes all elements of the stream.

In this thesis we focus on time-based windows, and our approach is extensible for tuple-based windows.

The window moves forwards as new tuples come. Time-based windows can move in two ways, tumbling and sliding [16]. Figure 2.1 gives an example for these two types of windows. A time-based sliding window each time moves a fixed length, which is defined as *window slide unit* (notated as U). A tumbling window can be considered as a sliding window with $U = W$.

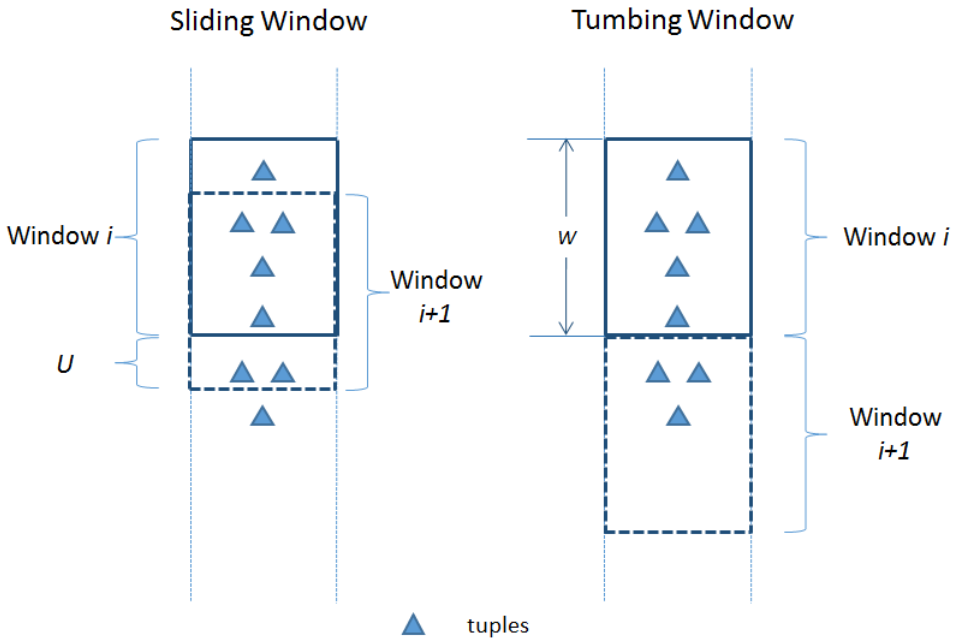


Figure 2.1: Sliding window versus tumbling window

Apply tumbling windows on two input streams with different window sizes, will result in two streams out-of-sync, because the stream with smaller window size always moves slower than the one with larger window size. In this thesis, we allow streams with different window sizes. Therefore, we discuss sliding windows.

We summarize important notations used in this thesis on Page 67.

Chapter 3

Related Work

In recent years, several works have addressed the disorder handling problem in data stream processing. In this chapter, we briefly introduce the ways that they handle the intra-stream disorder problem in Section 3.1, and the inter-stream disorder problem in Section 3.2. We then give a short summary in Section 3.3.

3.1 Intra-Stream Disorder Handling

Aiming to solve the out-of-order problem within an individual stream, a variety of approaches have been proposed. In this section, we demonstrate buffer-based approaches (Section 3.1.1), punctuation-based approaches (Section 3.1.2), speculation-based approaches (Section 3.1.3), and a hybrid approach combining buffering and speculation (Section 3.1.4).

3.1.1 Buffer-Based Approaches

We first look at the buffer-based disorder handling approaches. Buffer-based approaches use buffers to handle the disorder problem. We will take the out-of-order stream S in Figure 3.1 as input stream for all examples in this section. Tuples are notated as ei where i represents the timestamp of the tuple. For instance, $e1$ denotes a tuple with timestamp 1.

Figure 3.1: Stream S ($e1$ and $e5$ are out-of-order)

Static *K*-Slack Approach

Babu et al. [6] have introduced the concept of *k-constraints*, which is later often referred to as K-Slacks. A *slack* is a buffer that can temporarily store

and sort incoming tuples. The clk is defined as the largest timestamp that is ever observed in a stream. With a predefined parameter K, the slack stores all tuples with a timestamp within the time interval (clk-K, clk], and releases all tuples with a timestamp smaller than or equal to clk-K in ascending order of timestamps.

Example Figure 3.2 gives an example how the static K-Slack approach is used to handle the out-of-order Stream S. When out-of-order tuple $e1$ comes, clk is 3, the slack is buffering $e2$ and $e3$. Since $1 \notin (clk - K, clk]$, $e1$ does not go into the slack but goes directly to the output. Afterwards, $e4$ comes and updates clk to 4, so that $e2$ can be released. Similarly, the slack releases $e5$ when $e7$ comes. The output stream of K-slack is in-order.

Stream S	e2	e3	e1	e4	e6	e5	e7	e8	...
clk	2	3	3	4	6	6	7	8	...
clk-K	0	1	1	2	4	4	5	6	...
Buffer	{e2}	{e2,e3}	{e2,e3}	{e3,e4}	{e6}	{e5,e6}	{e6,e7}	{e7,e8}	...
Output			e1	e2	e3,e4		e5	e6	...

Figure 3.2: Static K-Slack Approach (K=2)

From this example we can see that, a K-Slack with a static size of 2 time units can tolerate tuples which are delayed by at most 2 time units. If $e1$ comes after $e4$ (delayed by 3 time units), then $e2$ will be before $e1$ even if in the output of K-Slack. As a concequence, it causes that the processing of each in-order tuple is delayed by at least 2 time units[1].

Adaptive K-Slack approach

In the static K-slack approach, the parameter K is chosen in advance and stays invariant during the processing. Therefore, a-priori knowledge of the incoming stream data is required. An improperly chosen K may lead to either a still-not-in-order output stream, if K is too small; or a huge result latency, if K is too large.

Aiming to overcome this drawback, Mutschler et al. [13] propose the adaptive K-slack approach, in which K is initialized to 0 and is dynamically adjusted

[1]Tuple $e3$ is delayed by 3 time units. This is because in the model of Babu et al., the slack can only "infer" the "current global time" from the largest observed timestamp(clk). And since $e5$ is late, it does not update clk until $e6$ comes.

according to the runtime measurement of the tuple delays ($clk - e.ts$). This means that no a-priori knowledge is required.

Example Figure 3.3 gives an example of the adaptive K-Slack approach. K is initialized to 0. When $e1$ comes, K is still 0 because there have been no out-of-order tuples. At this moment, an out-of-order $e1$ in the output of K-Slack is inevitable, because $e2$ and $e3$ have already been released. K is updated to 2, because $e1$ is late for 2 time units. Any future tuples late for no more than 2 time units will be tolerated, like $e5$. A future tuple which is late for more than 2 time units will still be an out-of-order tuple in the output of K-Slack and will update K in the meantime.

Stream S	e2	e3	e1	e4	e6	e5	e7	e8	...
K	0	0	2	2	2	2	2	2	...
clk	2	3	3	4	6	6	7	8	...
clk-K	2	3	1	2	4	4	5	6	...
Buffer				{e4}	{e6}	{e6}	{e6,e7}	{e7,e8}	...
Output	e2	e3	~~e1~~		e4	e5		e6	...

Figure 3.3: Adaptive K-Slack approach

Quality-Driven Adaptive *K*-Slack Approach

Considering that eventual 100% correctness of the result is not always required [2], Zhou [22] proposes the *quality-driven* adaptive K-Slack approach, which extends the original adaptive K-Slack approach and can trade the result correctness beyond user's expectation for lower result latency. Figure 3.4 shows the architecture of Zhou's system.

Besides K in the original adaptive K-Slack approach, Zhou uses an additional parameter α ($\alpha \in [0, 1]$), which reduces the buffer size from K to αK. Hence, the slack only buffers the tuples with a timestamp within the time duration (clk-αK, clk].

In this architecture, the K-Slack component not only buffers input tuples, but also tracks the result quality and passes it to the α monitor. Different adaptation algorithms like a PD controller [4] or a TCP controller [14] can be applied in the α monitor to adjust α according to the current and expected quality values. The updated α is returned to and later applied by the K-Slack component. Evaluation results show that a PD controller can achieve a

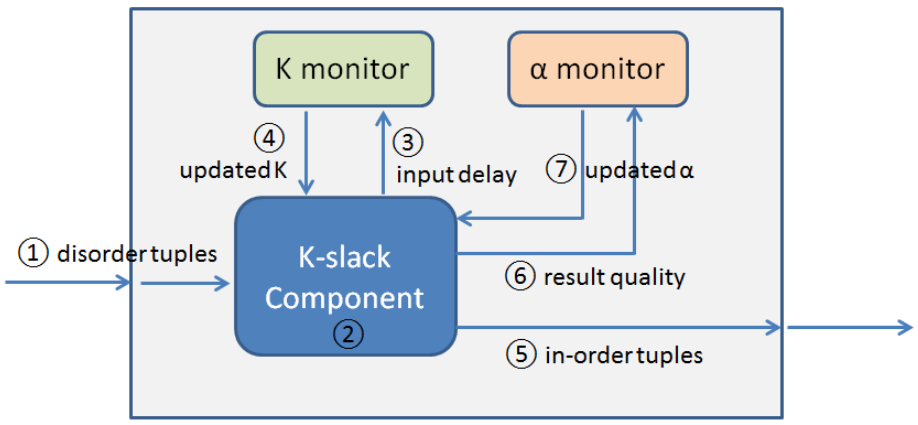


Figure 3.4: Internal structure of disorder handling component in [22]

good trade-off between result quality and result latency with proper parameter configuration of the PD controller. Zhou's work only supports aggregation operators currently.

3.1.2 Punctuation-Based Approaches

Tucker [20] first uses *punctuations* to deal with the disorder problem in data stream processing. Punctuations are special tuples embedded in a data stream that mark the end of substreams. Punctuations can be generated in different places, such like at data sources [20][2], at middlewares [11], or in the SPE [12]. Figure 3.5 shows an example, in which punctuations are provided by an input manager [11].

In this example, the SPE counts the number of tuples within a tumbling window with a size of 5 time units. This means it outputs the result at the end of every 5 time units. The column **Input Data** shows the original timestamps of tuples. The column **Control** shows the punctuation tuples generated by the input manager based on some heuristic. The SPE records counting result as **Count State** internally, and outputs it as **Output Tuple** once receiving a punctuation tuple.

More specifically, from Row 1, the SPE counts every input tuple. In Row 7, the punctuation with timestamp 5 indicates that the window $(0, 5]$ is end. Therefore, at this moment the SPE is safe to output a result (6,5) and reset the counter to 0. Since then, any Input Data tuple with a timestamp not larger than 5 (like in Row 9 or in Row 12) is discarded. In Row 13 comes another punctuation with timestamp 10, which ends the window $(5, 10]$.

RowNum	Input Data	Control	Count State	Output Tuple
1	1		1	
2	3		2	
3	2		3	
4	4		4	
5	2		5	
6	1		6	
7		5		(6, 5)
8	6		1	
9	~~2~~		1	
10	9		2	
11	7		3	
12	~~3~~		3	
13		10		(3, 10)
14	12		1	
15	~~8~~			
16	~~4~~			
17	~~3~~			
18	~~9~~			
19		15		(1, 15)

Figure 3.5: Punctuation-based disorder handling [11]

3.1.3 Speculation-Based Approaches

In speculation based approaches, the SPE assumes that input is in-order and does not wait for late arrivals. When a late tuple arrives, the system will invalidate early results which are affected by the late arrival, and produce an updated result by replaying the stream from the late tuple's timestamp to correct the result [8].

Example Figure 3.6 demonstrates an example using speculation-based approach. The SPE counts the number of tuples within a tumbling window with a size of 2 time units. When $e3$ arrives, the system will output the result $count([1,2]) = 1$, since the window [1,2] is over. When the out-of-order tuple $e1$ arrives, the SPE knows the previous result was wrong, hence reproduce a new correct result $count([1,2]) = 2$. Not all out-of-order tuples can result in replaying. For example, the out-of-order tuple $e5$ does not cause replaying, because it does not invalidate any previous results.

The speculation based approach is able to provide "early but conceivably wrong" results which are valuable in some contexts. It can also achieve eventual correctness with replaying. However, replaying requires additional storage and computation.

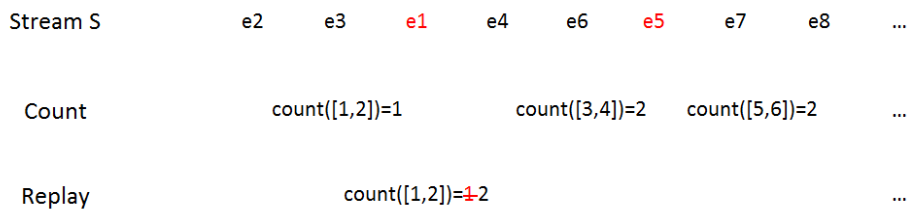


Figure 3.6: Speculation-based approach

3.1.4 Hybrid Approach

Experimental evaluations in [13] show that the latency is still large even though K is determined at runtime. Mutschler et al. [14] have proposed the speculative K-slack approach, which is a combination of the original adaptive K-slack approach discussed in Section 3.1.1, and the specutation-based approach discussed in Section 3.1.3.

Compared to the original adaptive K-slack approach, Mutschler et al. introduce an additional parameter $\alpha(\alpha \in [0, 1])$ to reduce the buffer size from K to αK. As a result, in speculative K-slack approach, the slack only buffers the tuples with a timestamp within the time duration $(clk\text{-}\alpha K,\ clk]$. Compared to the pure specutation-based approach, this buffer reduces the chance of replaying caused by late arrivals.

Example Figure 3.7 shows an example of speculative K-Slack approach with $\alpha = 0.5$. The only out-of-order tuple in the output of K-Slack is $e1$, which can cause potential replaying. Note that after $e5$, the output is one-time-unit advancing compared with the output in the adaptive K-Slack example, because the buffer size is decreased by $1(= (1 - \alpha)K)$ time unit in the speculative K-Slack approach.

Stream S	e2	e3	e1	e4	e6	e5	e7	e8	...
K	0	0	2	2	2	2	2	2	...
clk	2	3	3	4	6	6	7	8	...
clk-αK	2	3	2	3	5	5	6	7	...
Buffer				{e4}	{e6}	{e6}	{e7}	{e8}	...
Output	e2	e3	~~e1~~		e4	e5	e6	e7	...

Figure 3.7: Speculative K-Slack approach ($\alpha = 0.5$)

Taking advantages of speculation and replaying, the speculative K-slack approach can achieve eventual correctness of execution with a smaller buffer size thus lower latency, compared to adaptive K-slack approach. The shortcoming is also obvious. Additional storage is required to store released tuples in order to enable replaying. If α is chosen improperly small, replaying may be frequent which causes waste of computation as well as temporarily wrong results.

3.2 Inter-Stream Disorder Handling

In data stream processing, when we have operators like JOIN, UNION and INTERSECT, which take more than one streams as input, we must also take inter-stream disorder into consideration, because in a distributed topology it is a trivial situation. Figure 3.8 shows a typical moment when two streams are out-of-sync. Stream B is the leading stream at this moment. The out-of-sync tuples are those tuples that have larger timestamps than any tuple in the lagging stream A.

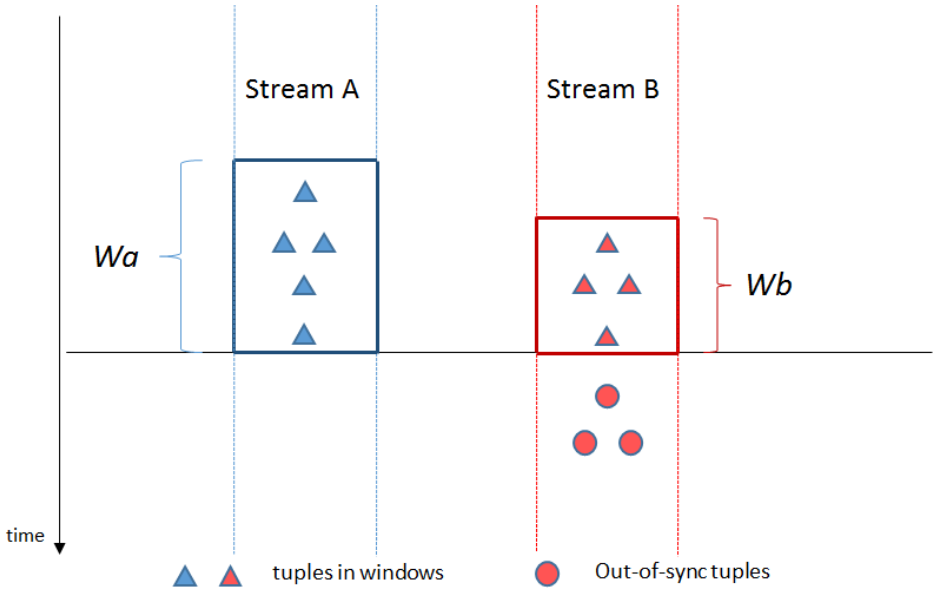


Figure 3.8: Out-of-sync streams

Several works have studied how to perform in-order execution over out-of-sync data streams. We describe the Sync-Filter approach, the Filter-Order approach, and deterministic merge in this section. Note for now we only focus on the inter-stream disorder problem, so in this section we assume that there is no intra-stream disorder, i.e., any single stream is in-order.

3.2.1 Sync-Filter and Filter-Order Approach

One straightforward approach, later being referred to as Sync-Filter approach [10], is to buffer the out-of-sync tuples, and produce result based on synchronized input streams at the SPE [19].

Hammad et al. [10] point out that in Sync-Filter approach, the production of result tuples are delayed due to synchronization. For example, in Figure 3.8, an out-of-sync tuple b in Stream B cannot be joined with any tuple in Stream A at this moment, even though they should be joined according to the semantic of the join query. They can only be joined after Stream A has a tuple a with $a.ts \geq b.ts$, so that b is no longer an out-of-sync tuple. This leads to the processing unit to be idle when streams are waiting for synchronization, and causes a huge overload when streams become synchronized, when all JOIN operations happen.

Hammad et al. propose the Filter-Order approach in order to overcome this drawback. In the Filter-Order approach, tuples can be joined when they arrive at the SPE. However, this will lead to an out-of-order output stream. Thus, an output buffer is used to temporarily store join results. Hammad et al. also give numerical analysis to the average response time of both Sync-Filter approach and Filter-Order approach under different input tuple arrival rates. A threshold is found so that the system is able to switch between Sync-Filter approach and Filter-Order approach to have a better trade-off between memory consumption and response time.

3.2.2 Deterministic Merge

Aguilera et al. [3] use *deterministic merge* algorithms to ensure multiple *merger* nodes in a decentralized publish/subscribe system to merge the message streams in exactly the same way, regardless of the delays of messages by the network.

As shown in Figure 3.9, several subscribers are assigned to one of the mergers. When publishing a message, a publisher first sends it to a logger to log it. Then the logger uses a FIFO link to send this message to any merger that has any subscriber interested it. The deterministic merge algorithm is replicated at every merger node. It can guarantee that all mergers order the messages consistently but independently. One simple deterministic merge algorithm can be that the producers attach timestamps to the messages, and then the mergers order the messages according to the timestamps in ascending order. Better algorithms are proposed in [3].

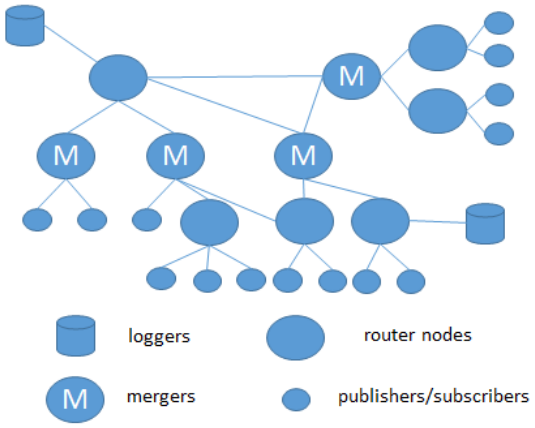


Figure 3.9: The architecture of a decentralized publish/subscribe system [3]

3.3 Summary

For intra-stream disorder handling, the static K-Slack approach and the adaptive K-Slack approach are good for their simplicity and effectiveness. However, they fail to adjust the buffer size dynamically, to have a trade-off between result quality and result latency according to users' expectation. The quality-driven approach with a PD controller is able to do so, but its performance highly depends on the configuration of parameters [22]. Moreover, it only supports aggregation operations currently. The performance of punctuation-based approaches also depends on the algorithm of generating punctuations. The speculation-based approaches can provide eventual correctness sacrificing additional storage and computation, yet a correct though very late result is arguably meaningless to some applications.

For inter-stream disorder handling, the approaches which are discussed in Section 3.2 all require that each input stream should be in-order. Moreover, there are some additional requirements such like in [3], each producer must be able to estimate all producers' expected message rates.

In our work, we consider more general situations for JOIN operation, where streams can be both out-of-order and out-of-sync. Therefore, we must face both intra-stream and inter-stream disorder problem at the same time. We also wish to have a quality-driven approach, so that we can provide result that can meet users' requirement on quality with minimum result latency. We demonstrate our approach in the next chapter.

Chapter 4

System Model

In this chapter, we demonstrate the system model for sliding window JOIN upon data streams. We first have an overview of our solution, then we explain our join logic in detail.

4.1 Solution Overview

In order to handle the disorder problem and give an in-order result stream for JOIN operation, we adopt a similar system architecture proposed in [21].

As shown in Figure 4.1, one K-Slack is applied to each input stream. Their sizes are set to 0 initially and are adjusted at runtime (see Section 5.4). Since the output streams of K-Slacks can be potentially out-of-sync, we take the Sync-Filter approach as explained in Section 3.2.1. A synchronization buffer *SyncBuffer* is used to buffer early tuples in the leading stream in order to wait for tuples in the lagging stream. The *SyncBuffer* is actually a pair of buffers storing early tuples from Stream A and Stream B. We use *SyncBufA* and *SyncBufB* to denote them. The window of the leading side receives synchronized tuples from the *SyncBuffer*, and the window of the lagging side receives tuples directly from the input stream.

Figure 4.2 shows the window view at a certain moment. Two windows, the lagging WindowA with size W_A, and the leading WindowB with size W_B, have the same end point, which equals the smaller one of both streams' maximum timestamp of all arrived tuples. We denote the current common endpoint of both windows as $WindowEnd$. When $WindowEnd$ moves forward, two windows move at the same time. Tuples that have already arrived with timestamps greater than $WindowEnd$ will be pending in *SyncBufA* or *SyncBufB*. There can be at most one nonempty buffer at any point of time. If both buffers are nonempty, $WindowEnd$ will move forward until one buffer is empty.

For a *sliding* window, it cannot move if we have not seen not a pending tuple

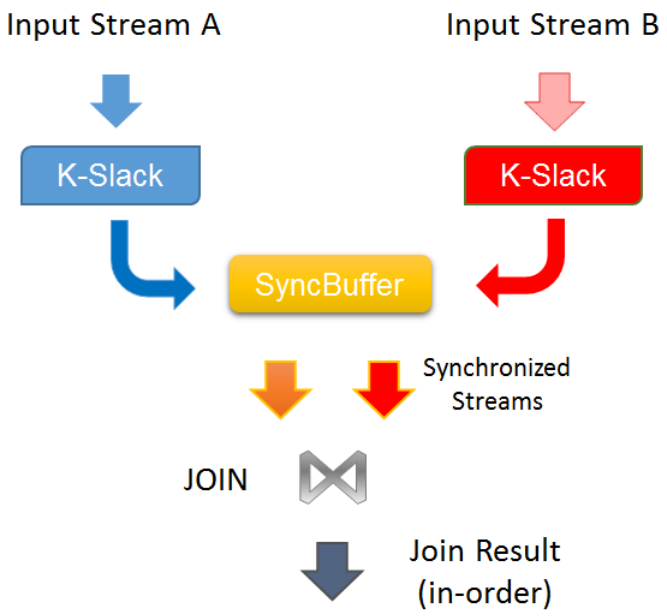


Figure 4.1: Sliding-window JOIN with disorder handling

e with $e.ts \geq WindowEnd + U$. So there is a possible counterexample that both windows have pending tuples but $WindowEnd$ does not move, because the windows are not able to slide. For general-case sliding windows, we always wait until the window is able to slide. Right now, for simplicity, we assume that U equals the smallest time unit.

4.2 Join Logic

We now explain the detailed join logic.

$JOIN(a, b)$ is the JOIN operation applied upon a (a tuple in Stream A) and b (a tuple in Stream B). If the join condition is fulfilled, $JOIN(a, b)$ returns an result tuple r with $r.ts = max(a.ts, a.ts)$; otherwise it returns no result tuple.

Since we must guarantee that the result stream is in-order, and there is no extra disorder handling on the result stream, a result tuple r can be released iff $r.ts = WindowEnd$, that is to say, they can only be released when their the timestamp equal to the current end point of the windows. We achieve this by dropping out-of-date result tuples and not generating result tupls ahead of $WindowEnd$. First, if a result tuple r has $r.ts < WindowEnd$ when it is generated, it must be dropped. Otherwise it will lead to the result stream R to be out-of-order. Second, no result tuple r with $r.ts > WindowEnd$ is generated. The JOIN operator waits until $WindowEnd$ moves to $WindowEnd' = r.ts$, and then generate it.

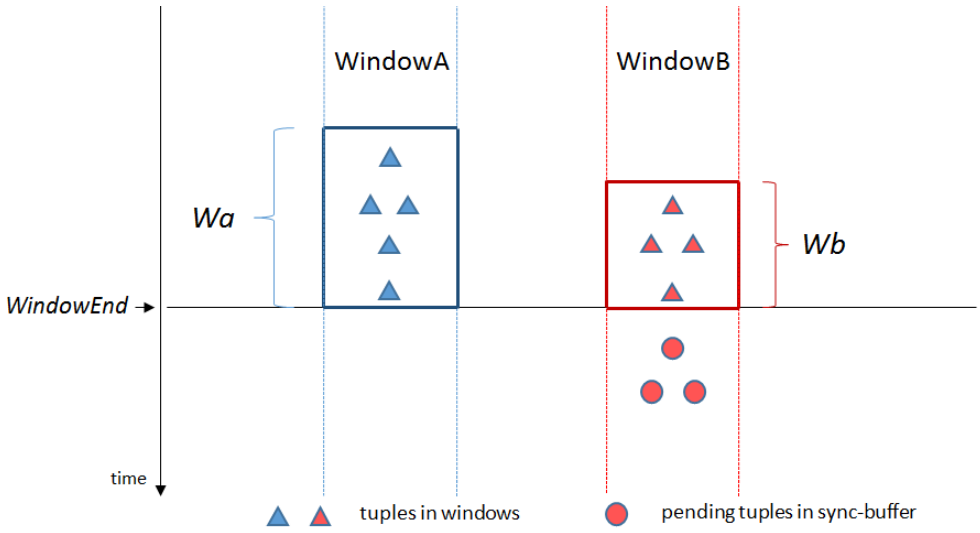


Figure 4.2: Sliding-window JOIN with sync-buffer

From the JOIN operator's perspective, it receives tuples from two K-Slacks, which may still be out-of-order due to insufficient buffer size. There are different situations for a newly received tuple e. Without losing generality, in this section we assume that tuple e is from Stream A.

Situation 1 $e.ts \leq WindowEnd - W_A$

In this case, e is too late to be able to contribute to any result tuples r with $r.ts \geq WindowEnd$, so e is simply dropped.

Situation 2 $WindowEnd - W_A < e.ts < WindowEnd$

In this case, e arrives late but its time stamp is till within the current scope of WindowA, so it is still able to contribute to result tuples r with $r.ts \geq WindowEnd$. The SPE joins e with all tuples $b \in WindowB$ with $b.ts = WindowEnd$, and stores e in WindowA.

Situation 3 $e.ts = WindowEnd$

In this situation, e is not an out-of-order tuple. Hence, it will be joined with all tuples $b \in WindowB$, and then be stored in WindowA.

We demonstrate situation 1, situation 2 and situation 3 in Figure 4.3.

Situation 4 $e.ts > WindowEnd$. There are three sub-situations as follows:

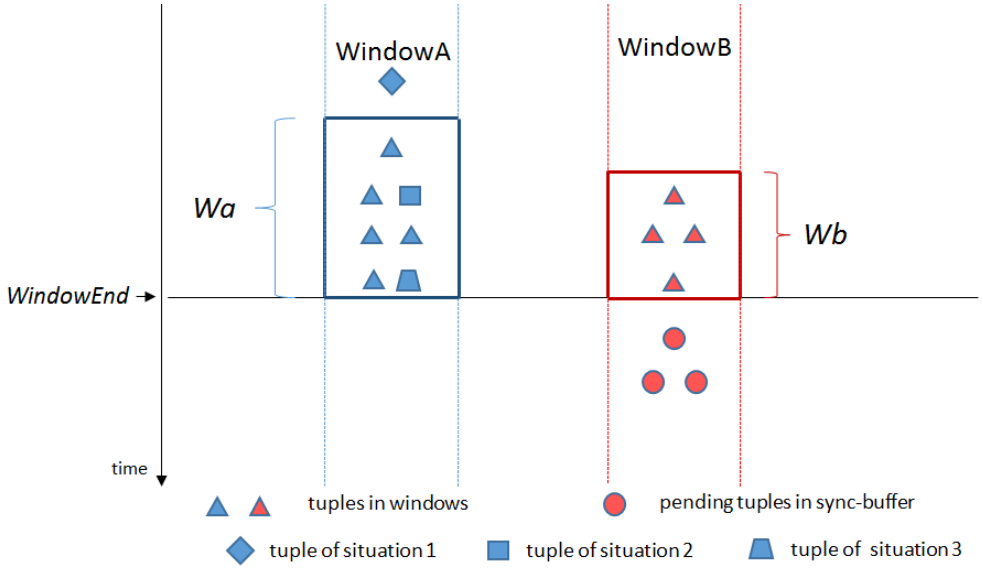


Figure 4.3: JOIN Logic, different situations

Situation 4a $e.ts > WindowEnd \wedge SyncBufB = \varnothing$.

$SyncBufB = \varnothing$ means that Stream B is not the current leading stream (Figure 4.4). e is simply inserted into $SyncBufA$. At this moment e will not be joined with any tuples, because any result tuple with e as a component will have a timestamp of at least $e.ts$, which is greater than the current $WindowEnd$.

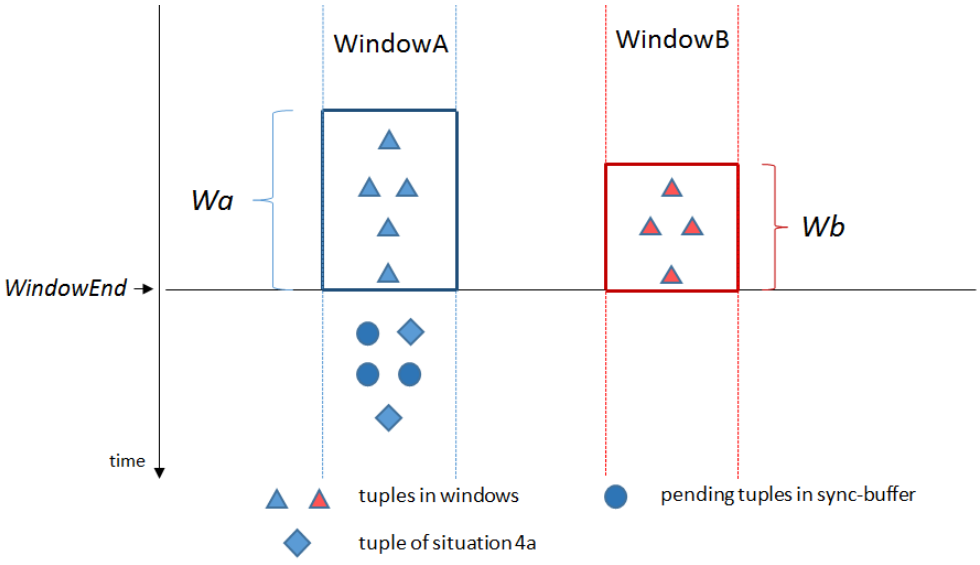


Figure 4.4: JOIN Logic, Situation 4a

Situation 4b $e.ts > WindowEnd \wedge SyncBufB \neq \varnothing \wedge e.ts \leq max(b.ts), b \in SyncBufB$

$SyncBufB \neq \varnothing$ means that Stream B is the current leading stream. $e.ts \leq max(b.ts), b \in SyncBufB$ means that Stream B will still be leading (Figure 4.5). In this situation, $WindowEnd$ will be pushed forward, from the smallest timestamp of tuples in $SyncBufB$ in ascending order, until $WindowEnd$ reaches $e.ts$ Each movement of $WindowEnd$ triggers a series of actions:

> 1) each window will purge old tuples according to the new $WindowEnd$;
> 2) newly added tuples to the window will be joined with corresponding tuples.

Tuple e is the last tuple added to WindowA.

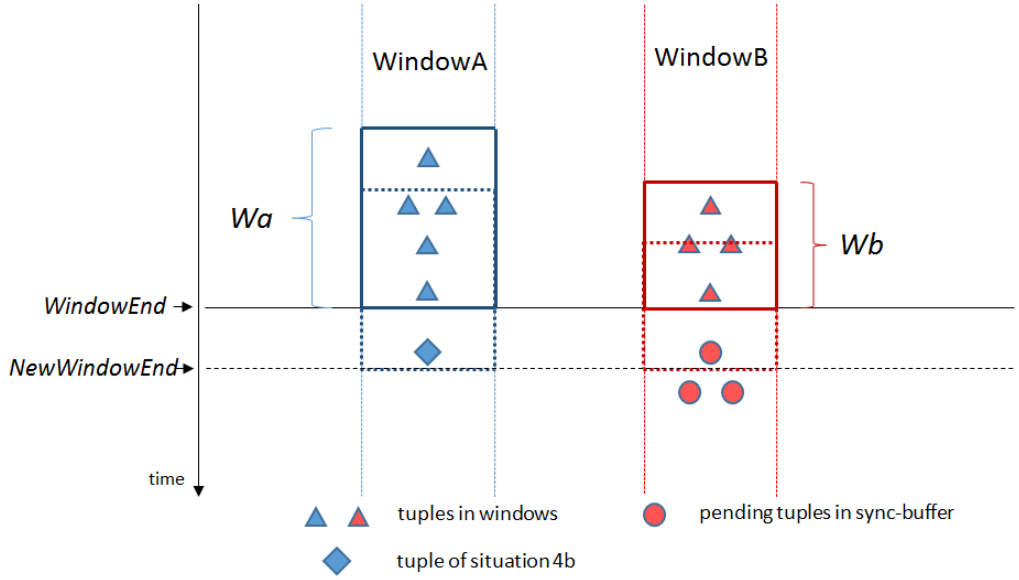


Figure 4.5: JOIN Logic, Situation 4b

Situation 4c $e.ts > WindowEnd \wedge SyncBufB \neq \varnothing \wedge e.ts > max(b.ts), b \in SyncBufB$

Stream B is the current leading stream, but it will no longer be leading after the window moves. Stream A will become the leading stream, because $e.ts > max(b.ts), b \in SyncBufB$ (Figure 4.6). The windows move in a similar way as in **Situation 4b**, but the new $WindowEnd$ will be $max(b.ts), b \in SyncBufB$, and there will be no tuple left in $SyncBufB$. Stream A becomes the leading stream, and e is inserted into $SyncBufA$.

We present the pseudo code of the JOIN logic in Algorithm 1. In a word, result tuples can only be tagged with the timestamp of the current $WindowEnd$,

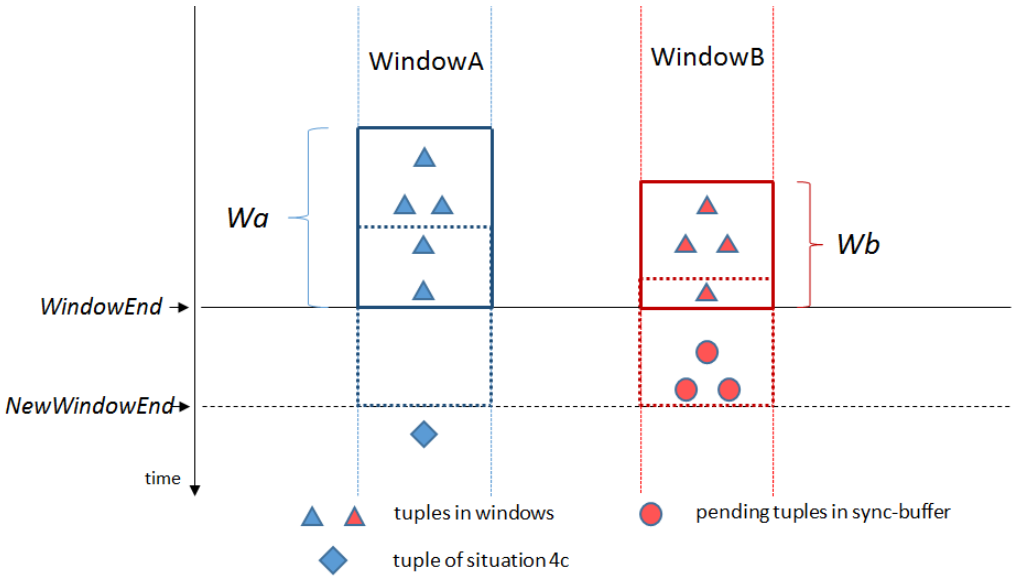


Figure 4.6: JOIN Logic, Situation 4c

because we want the result stream to be in-order, and there is no K-Slack on the result stream, as we mentioned in the beginning of this section.

Algorithm 1: Join Logic

Input: StreamA, StreamB
Output: StreamR
$WindowA = \varnothing; WindowB = \varnothing;$
$SyncBufA = \varnothing; SyncBufB = \varnothing; WindowEnd = 0$
while $e \leftarrow getIncomingTuple()$ **do**
  if $e \in StreamA$ **then**
    if $e.ts \leq WindowEnd - W_A$ **then**
      drop e; ▷ Situation 1
    else if $e.ts \leq WindowEnd$ **then**
      $R \leftarrow$ join e with all $b \in WindowB$;
      output $r \in R$, if $r.ts = WindowEnd$, to StreamR;
      WindowA.insert(e); ▷ Situation 2,3
    else if $SyncBufA \neq \varnothing$ **then**
      SyncBufA.insert(e); ▷ Situation 4a
    else
      while $\exists b \in SyncBufB, b.ts \leq e.ts$ **do**
        $b \leftarrow SyncBufB.popTupleWithMinTS()$;
        $WindowEnd \leftarrow b.ts$;
        delete $a \in WindowA$, if $a.ts \leq WindowEnd - W_A$;
        $R \leftarrow$ join b with all $a \in WindowA$;
        output $r \in R$ to StreamR;
        WindowB.insert(b); ▷ lazy retention here
      if $SyncBufB \neq \varnothing || e.ts == WindowEnd$ **then**
        $WindowEnd \leftarrow e.ts$;
        delete $a \in WindowA$, if $a.ts \leq WindowEnd - W_A$;
        delete $b \in WindowB$, if $b.ts \leq WindowEnd - W_B$;
        $R \leftarrow$ join e with all $b \in WindowB$;
        output $r \in R$ to StreamR;
        WindowA.insert(e); ▷ Situation 4b
      else
        delete $b \in WindowB$, if $b.ts \leq WindowEnd - W_B$;
        SyncBufA.insert(e); ▷ Situation 4c
  else
    do similar; ▷ $e \in StreamB$

Chapter 5

Quality-Driven Slack Size Adaptation

In this chapter, we demonstrate our approach to adjust the K-Slack buffer size in order to meet the users' requirement on the join result quality. In Section 5.1, we introduce the formal definition of result quality. In Section 5.2, we introduce the concept of late degree distribution. In Section 5.3, we reveal the relationship among late degree distributions, buffer sizes and the result quality. We also manage to estimate the result quality if the other two are known. In Section 5.4, we discuss the detailed slack size adaptation approach based on the conclusion of Section 5.3.

5.1 Quality Metric

It is important to have a quantitative indicator to demonstrate how good a join result set is. According to Algorithm 1 in Chapter 4, late arrived input tuples can cause missing of result tuples. However, they do not have any side effect on the other results. In other words, there are never "bad results" produced because of late arrived input tuples. Therefore, the quality of a join result set over a period of time can be defined as the number of result tuples that are *actually produced* within this period of time (notated as $Nr_{produced}$), divided by the number of result tuples that *should be produced* within this period of time, if there is no disorder in input streams (notated as $Nr_{original}$). Formally, we define the result quality of a join result set, denoted by Q, as in [21]:

$$Q = \frac{Nr_{produced}}{Nr_{original}} \times 100\% \tag{5.1}$$

In general, the fewer missing result tuples we have, the better the result quality we achieve. When no input tuple arrives late, the number of result tuples we actually produce is equal to the number of original results, so the quality would be 100%. Since by no means can we produce more result tuples than the complete result set, the quality can never surpass 100%.

5.2 Late Degree Distribution

We use a late degree distribution D to describe the disorder pattern of an out-of-order stream within a time interval. A late degree distribution D is a probability distribution, describing the pattern of arrival times of input tuples in the stream. The late degree distribution of a stream S can be denoted as:

$$D_S = \{p_{S0}, p_{S1}, p_{S2}, ..., p_{S\Lambda_{max}}\}, sum(p_{S\Lambda}) = 1$$

Λ represents the late degree of an input tuple. If an input tuple e has a timestamp $e.ts$ not smaller than the largest timestamp that has been processed when e arrives (notated as $e.ts_{process}$), then $\Lambda = 0$; if e has a timestamp $e.ts$ smaller than $e.ts_{process}$, then it has a late degree $\Lambda = e.ts_{process} - e.ts$. For a single stream with no K-Slack, $e.ts_{process}$ equals the maximum observed timstamp clk when e arrives. For the join logic we proposed in Algorithm 1, $e.ts_{process}$ equals the $WindowEnd$ when e arrives.

$p_{S\Lambda}$ represents the probability that a tuple has a late degree of Λ. Exceptionally, $p_{S\Lambda_{max}}$ represents the probability that a tuple has a late degree of Λ_{max} or more. The sum of all $p_{S\Lambda}$ should be 1. We can observe empirical probabilities for all Λ by recording Λ of each tuple before the K-Slack (see Section 6.1.1 for detail), thereby estimate each value in D_S.

Figure 5.1 shows a possible late degree distribution $D_S = \{p_{S0} = 0.6, p_{S1} = 0.2, p_{S2} = 0.1, p_{S3} = 0.1\}$. Based on this distribution we can say that, for an input tuple e in Stream S, the probability that it arrives on time (with $e.ts$ equal to the clk **of the moment it arrives**) is 60%; the probability that it is delayed by 1 time unit (with $e.ts = clk - 1$) is 20%; the probability that it is delayed by 2 time units (with $e.ts = clk - 2$) is 10%; the probability that it is delayed by not less than 3 time units (with $e.ts \geq clk - 3$) is 10%.

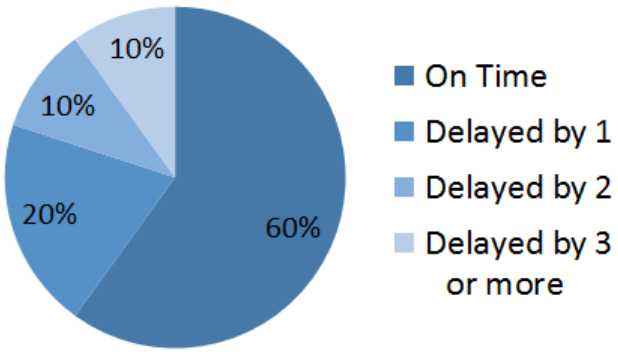


Figure 5.1: Example of late degree distribution D_S

If D_S is considered constant within a short period of time (in this example 4

time units), we can go one step further, deriving that for an input tuple e in Stream S, the probability that it arrives before clk reaches $e.ts+1$ is 60%; the probability that it arrives before clk reaches $e.ts+2$ is 80%(=60%+20%); the probability that it arrives before clk reaches $e.ts$+3 is 90%(=60%+20%+10%). (Figure 5.2)

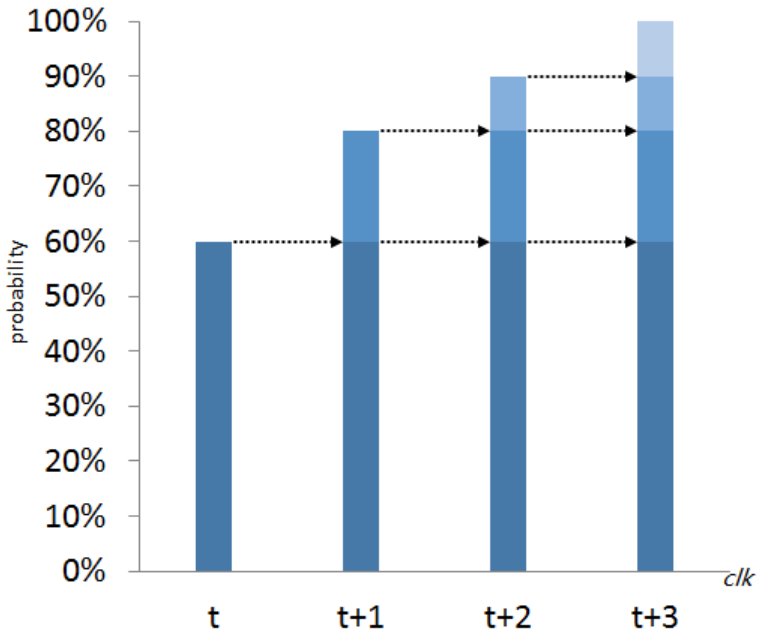


Figure 5.2: Arrived time probability

Now we consider reversely. At the moment that clk increases from t to $t+1$, for a tuple e with $e.ts = t$, the probability that e has arrived is 60%; for e with $e.ts = t-1$, the probability that e has arrived is 80%; for e with $e.ts = t-2$, the probability that e has arrived is 90%.

5.2.1 Transformation of Late Degree Distributions

Applying K-Slack on the input stream will change the late degree distribution. For example, consider that applying a static K-Slack with a size of 1 time unit. This will allow all in-order input tuples delayed by 1 time unit to be reordered by the K-slack buffer and output in correct order (see Section 3.1.1), which means for all input tuples e, we have $e.ts'_{process} = e.ts_{process} - 1$. Hence,

a) For those tuples that had $\Lambda = 0$, they will remain on time;

b) For those tuples that had $\Lambda > 0$, they will have a late degree after K-Slack Λ' with $\Lambda' = \Lambda - 1$.[1]

[1]Since $\Lambda = e.ts_{process} - e.ts$, and $\Lambda' = e.ts'_{process} - e.ts = e.ts_{process} - 1 - e.ts = \Lambda - 1$.

Therefore, the late degree distribution (shown in Figure 5.3) for the output stream of the aforementioned static K-slack with a size of 1 time unit, denoted as D'_S, can be calculated as:

$$D'_S(K=1) = \{p'_{S0} = p_{S0} + p_{S1} = 0.8, p'_{S1} = p_{S2} = 0.1, p'_{S2} = p_{S3} = 0.1\}$$

More generally, we could derive the late degree distribution after applying a K-Slack with a size of K time units:

$$D'_S(K) = \{p'_{S0} = \sum_{i=0}^{K} p_{Si}, p'_{S\Lambda} = p_{S(K+\Lambda)}, ..., p'_{S(\Lambda_{max}-1)} = p_{S\Lambda_{max}}\}, \quad (0 < \Lambda \leq \Lambda_{max} - K) \tag{5.2}$$

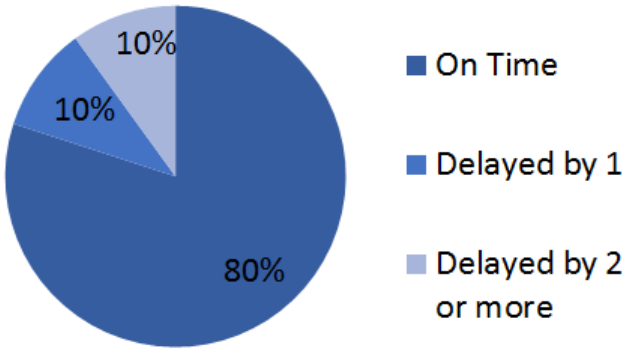


Figure 5.3: Late degree distribution D'_S of the output stream of K-Slack

Similarly, *SyncBuffer* also changes the late degree distribution, since applying a *SyncBuffer* with a size of L time units will also make the processing of all in-order input tuples delayed by L time units, which means for all input tuples e, we have $e.ts'_{process} = e.ts_{process} - L$. In general, if an out-of-order stream S has a K-Slack with size K and a *SyncBuffer* with size L, its original late degree distribution $D_S = \{p_{S0}, p_{S1}, p_{S2}, ..., p_{S\Lambda_{max}}\}$ will be transformed to:

$$D'_S(K, L) = \{p'_{S0} = \sum_{i=0}^{K+L} p_{Si}, p'_{\Lambda} = p_{S(K+L+\Lambda)}, ..., p'_{S(\Lambda_{max}-1)} = p_{S\Lambda_{max}}\}, \quad (0 < \Lambda \leq \Lambda_{max} - K - L) \tag{5.3}$$

5.3 Quality Estimation

Now we present how we estimate the result quality of the JOIN operation within a time interval T which contains n time units. If T is short enough, we can consider that the data rates and the late degree distributions of both streams, as well as the join selectivity are stable (see discussions in Section 5.3.3). We denote the number of tuples with each timestamp in both streams as n_A or n_B, and the join selectivity as ρ.

To help explain the key idea of our solution, we give an example, in which we join two streams A and B with window slide unit $U = 1$ time unit. To ease the presentation, we assume that both streams have the same late degree distribution D as shown in Figure 5.1. We also assume that the two streams are in-sync, and no K-Slack is applied. We will relax these assumptions and discuss the general case in Section 5.3.4.

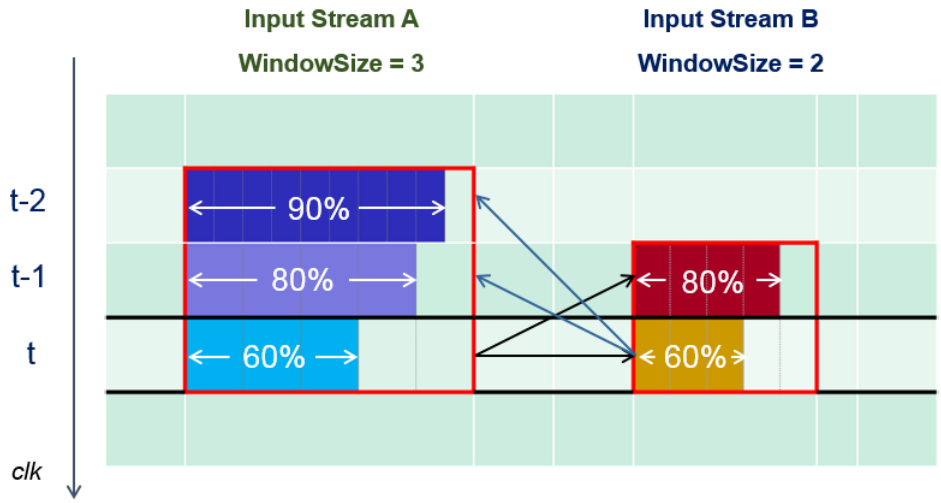


Figure 5.4: Quality estimation for t

5.3.1 Estimation of $Nr_{original}$

We now demonstrate how we estimate $Nr_{original}(t)$ for any time unit t in T.

A result tuple r with timestamp $r.ts = t$ can be produced as a result of either a tuple a from Stream A with $a.ts = t$, or a tuple b from Stream B with $b.ts = t$, or both of them. Therefore, $Nr_{original}(t)$ contains four parts:

Part 1. a with $a.ts = t$ JOIN b with $b.ts = t$;

Part 2. a with $a.ts = t$ JOIN b with $b.ts = t - 1$;

Part 3. a with $a.ts = t - 1$ JOIN b with $b.ts = t$;

Part 4. a with $a.ts = t - 2$ JOIN b with $b.ts = t$.

In Figure 5.4, the red boxes represent two windows. The height of each red box represents the window size, and the width represents the data rate of each stream. There are four arrows between the two boxes, representing the four parts mentioned above. For the time unit t, we have:

$$Nr_{original}(t) = n_A \times n_B \times 4 \times \rho \tag{5.4}$$

Applying 5.4 to all n time units within T, we can calculate $Nr_{original}$ for T:

$$Nr_{original} = n_A \times n_B \times 4 \times \rho \times n \tag{5.5}$$

5.3.2 Estimation of $Nr_{produced}$

We now look at how we estimate the expected value of $Nr_{produced}(t)$ for any time unit t in T.

Clearly, we can count the actually produced join results, thus $Nr_{produced}$, after execution. However, since we want to control the result quality by adjusting the slack size, it is also important to estimate the $Nr_{produced}$ based on known late degree distributions beforehand.

Similar to $Nr_{original}(t)$, $Nr_{produced}(t)$ also contains four parts. Again in Figure 5.4, the percentages show the probabilities that tuples with certain timestamps have arrived, so the width of the colored block containing such a percentage represents the estimated expected number of arrived tuples with certain timestamps. For example, at t, the estimated expected number of arrived tuples with timestamp $t-1$ are $n_A \cdot 80\%$ and $n_B \cdot 80\%$ for Stream A and B respectively.

Hence, for the time unit $t \in T$, we estimate the expected number of actually produced result tuples $E(Nr_{produced}(t))$ as:

$$\begin{aligned} E(Nr_{produced}(t)) &= (n_A \cdot 60\% \cdot n_B \cdot 60\% + n_A \cdot 60\% \cdot n_B \cdot 80\% + n_A \cdot 80\% \cdot n_B \cdot 60\% \\ &\quad + n_A \cdot 90\% \cdot n_B \cdot 60\%) \times \rho \\ &= n_A n_B \cdot 186\% \times \rho \end{aligned} \tag{5.6}$$

We apply Equation 5.6 to all n time units within T, and calculate the expected number of actually produced result tuples for T (denoted by $E(Nr_{produced})$):

$$E(Nr_{produced}) = n_A \cdot n_B \cdot 186\% \times \rho \times n \tag{5.7}$$

5.3.3 Estimation of the Result Quality

We now demonstrate how we estimate result quality Q_{est} of join.

According to Equation 5.1, along with Equation 5.5 and 5.7, we can calculate the expected result quality $E(Q)$:

$$E(Q) = \frac{E(Nr_{produced})}{Nr_{original}} = 46.5\% \tag{5.8}$$

According to the law of large numbers [17], if we have a large number of result tuples produced in T, then the actual quality Q of the join results should be close to the expected value of itself. Therefore, we can estimate the quality as:

$$Q_{est} = E(Q) = 46.5\% \tag{5.9}$$

Discussions

At the beginning of this section, we assume that the data rates and the late degree distributions of both streams, as well as the join selectivity are stable. All the derivations and calculations afterwards are based on this assumption. If these values have fluctuations, the estimation will be inaccurate. And the longer T is, the more fluctuations these values have, hence the more inaccurate the estimation will be.

However, in the last part of the estimation, we can only estimate Q_{est} as $E(Q)$ with a large $Nr_{produced}$. Generally, the longer T is, the larger $Nr_{produced}$ we get.

Therefore, the selection of the size of T is important to our approach, since either a too large or a too small T will influence the estimation of result quality largely. We evaluate different values of T in Section 6.3.2.1.

5.3.4 Generalization

Now we discuss the general case, where Stream A has an original late degree distribution D_A, and Stream B has an original late degree distribution D_B. K-Slacks with sizes of K_A and K_B are applied on these two streams respectively. The output streams of K-Slacks might be out-of-sync, so that either *SyncBufA* with size L_A or *SyncBufB* with size L_B exists (see Section 4.1).

First, we transform D_A and D_B according to Equation 5.3 and get:

$$D'_A(K_A, L_A) = \{p'_{A0} = \sum_{i=0}^{K_A+L_A} p_{Ai}, p'_{A\Lambda_A} = p_{A(K_A+L_A+\Lambda_A)}, ..., p'_{A(\Lambda_{max_A}-1)} = p_{A\Lambda_{max_A}}\},$$
$$D'_B(K_B, L_B) = \{p'_{B0} = \sum_{i=0}^{K_B+L_B} p_{Bi}, p'_{B\Lambda_B} = p_{B(K_B+L_B+\Lambda_B)}, ..., p'_{B(\Lambda_{max_B}-1)} = p_{B\Lambda_{max_B}}\},$$
$$(0 < \Lambda_A \leq \Lambda_{max_A} - K_A - L_A, 0 < \Lambda_B \leq \Lambda_{max_B} - K_B - L_B, L_A \cdot L_B = 0) \tag{5.10}$$

Then, generalizing Equation 5.6, we get:

$$E(Nr_{produced}(t)) = n_A n_B \cdot (p'_{A0} \cdot p'_{B0} + p'_{A0} \cdot \sum_{i=1}^{W_B} \sum_{j=0}^{i} p'_{Bj} + p'_{B0} \cdot \sum_{i=1}^{W_A} \sum_{j=0}^{i} p'_{Aj}) \times \rho \tag{5.11}$$

Following the derivation in Equation 5.7, Equation 5.8 and Equation 5.9, the estimated quality for the time interval T is calculated as:

$$Q_{est} = \frac{p'_{A0} \cdot p'_{B0} + p'_{A0} \cdot \sum_{i=1}^{W_B} \sum_{j=0}^{i} p'_{Bj} + p'_{B0} \cdot \sum_{i=1}^{W_A} \sum_{j=0}^{i} p'_{Aj}}{W_A + W_B - 1} \tag{5.12}$$

5.4 Buffer Size Adaptation Strategy

In this section, we demonstrate how we adjust buffer sizes to meet user's expected result quality Q_{exp} for the next time interval T, based on observed late degree distributions.

Literally, the goal of quality-driven slack size adaptation is, to obtain the result quality of user's requirement with minimum result latency. We define the latency λ of a single result tuple r as the difference between its timestamp and the maximum timestamp of observed input tuples when it is produced, i.e., if $r = JOIN(a, b)$, then

$$\lambda_r = max(clk_A, clk_B) - max(a.ts, b.ts) \tag{5.13}$$

For a large number of result tuples r, we can consider that the average timestamp of a approaches the average timestamp of b, i.e., $\overline{a.ts} = \overline{b.ts}$. Without losing generality, we assume that Stream A is not the lagging stream, i.e., $clk_A \geq clk_B$, then we have the average result latency $\overline{\lambda_r}$:

$$\overline{\lambda_r} = clk_A - \overline{a.ts} \tag{5.14}$$

which can be derived to:

$$\overline{\lambda_r} = K_A + L_A \tag{5.15}$$

According to Algorithm 1 in Chapter 4, there is no way to adjust L directly, since it is naturally resulted from the difference between the maximum observed timestamps of the output of the slack applied for Stream A and the maximum observed timestamps of the output of the slack applied for stream B. Namely,

$$L = |(clk_A - K_A) - (clk_B - K_B)| \tag{5.16}$$

(a) When $clk_A - K_A > clk_B - K_B$, *SyncBuffer* is at the side of Stream A, i.e., $L_A = (clk_A - K_A) - (clk_B - K_B)$, and $L_B = 0$. Equation 5.15 can be derived to:

$$\overline{\lambda_r} = clk_A - clk_B + K_B \tag{5.17}$$

(b) When $clk_A - K_A \leq clk_B - K_B$, *SyncBuffer* is not at the side of Stream A (either at the side of Stream B, or there is no *SyncBuffer*), i.e., $L_A = 0$, $L_B = (clk_B - K_B) - (clk_A - K_A)$. Equation 5.15 can be derived to:

$$\overline{\lambda_r} = K_A \tag{5.18}$$

To summarize (a) and (b), we have:

$$\overline{\lambda_r} = \begin{cases} clk_A - clk_B + K_B & \text{if } K_A < clk_A - clk_B + K_B; \\ K_A & \text{if } K_A \geq clk_A - clk_B + K_B. \end{cases} \tag{5.19}$$

Now, the problem of quality-driven slack size adaptation can be translated to: **given the current clk_A, clk_B, D_A and D_B, searching for a value set of K_A and K_B with minimum $\overline{\lambda_r}$, which transfers D_A and D_B to D'_A and D'_B, so that $Q_{est} \geq Q_{exp}$ in the near future.**

In order to predict the near future[2] from the history, we assume that the out-of-order degree $clk_A - clk_B$ and the late degree distributions D_A, D_B are roughly equivalent to their current value.

Starting from the very = beginning of the join processing, where K_A and K_B are set to 0. There are three possible situations:

Situation 1: $Q_{est} = Q_{exp}$
In this case, there is no need to adjust K_A and/or K_B;

Situation 2: $Q_{est} > Q_{exp}$
In this case, we do not adjust K_A and/or K_B, since we cannot set them to negative values. We also do not disable *SyncBuffer*, since it will decrease the

[2]The concrete length can be specified in our implementation. See Section 6.1.2.

result quality dramatically, according to our experiment result in Section 6.3 (`Baseline3`);

Situation 3: $Q_{est} < Q_{exp}$
In this case, we need to increase K_A and/or K_B. Let us denote the new values of K_A and K_B after the adaptation as K'_A and K'_B, respectively. We analyze the behavior of the streams after K-Slacks, namely Stream A' and Stream B':

> 3a) Increase K_A by δ. According to Equation 5.19, increasing K_A will not lead to a larger result latency $\overline{\lambda_r}$, unless K'_A surpasses $clk_A - clk_B + K_B$. Moreover, adding δ to K_A will make Stream A' less leading to Stream B', i.e., $L'_A = L_A - \delta$. it will not influence Q_{est}, when $K_A < clk_A - clk_B + K_B$. Because as long as Stream A' keeps leading, $K'_A + L'_A$ keeps equal to $K_A + L_A$.
>
> If K_A surpasses $clk_A - clk_B + K_B$, Stream A' is no longer leading. We discuss the situation of increasing K for the not leading stream in 3b.
>
> 3b) Increase K_B by δ. Since Stream B is not the leading stream, adding δ to K_B will make it even more lagging, so that L_A will be increased to $L_A + \delta$. Hence, we will have:
>
> ① $\overline{\lambda_r}$ is increased by δ;
>
> ② for Stream A, K_A does not change but L_A is increased by δ;
>
> ③ for Stream B, K_B is increased by δ, L_B is still 0.
>
> 3c) Increase both K_A and K_B by δ. We have:
>
> ① $\overline{\lambda_r}$ is increased by δ;
>
> ② for Stream A, K_A is increased by δ, L_A remains unchanged;
>
> ③ for Stream B, K_B is increased by δ, L_B is still 0.

For the cases 3b and 3c, when $\overline{\lambda_r}$ is increased by the same amount δ, according to Equation 5.10 and Equation 5.12, Q_{est} will be increased by the same amount.

To conclude, the cases 3b and 3c are equivalent if we need to increase Q_{est}. In our implementation, we take the approach of 3c. At any time of the join execution, we can use Algorithm 2 to estimate the minimum K_A and K_B we need to have for the next time interval T, in order to fulfill the user's requirement on the result quality.

Algorithm 2: Slack Size Adaptation

Input: D_A, D_B, Q_{exp}, clk_A, clk_B
Output: K_A, K_B
if $clk_A < clk_B$ **then**
  $L_A \leftarrow 0$; $L_B \leftarrow clk_B - clk_A$;
else
  $L_B \leftarrow 0$; $L_A \leftarrow clk_A - clk_B$;
$K_A \leftarrow 0$; $K_B \leftarrow 0$;
transform D_A and D_B according to Equation 5.10;
estimate Q_{est} according to Equation 5.12;
do
  $K_A \leftarrow K_A + k\_granularity$; ▷ See Section 6.1.2
  $K_B \leftarrow K_B + k\_granularity$;
  transform D_A and D_B according to Equation 5.10;
  estimate Q_{est} according to Equation 5.12;
while $Q_{est} < Q_{exp}$;

Chapter 6

Implementation and Evaluation

In this chapter, we first present the implementation detail of our proposal in Section 6.1. In Section 6.2 we describe the hardware and software configuration and the datasets that we use as input in our experimental evaluations. Besides, we also briefly introduce the PD controller algorithm and the baselines, which we use to compare with our disorder handling algorithm. In Section 6.3, we present the results of various experiments.

6.1 Implementation

We extend SAP® Sybase® Event Stream Processor (ESP)[18] for data stream processing. We introduce the components that we develop based on ESP in Section 6.1.1, and the parameters of our approach in Section 6.1.2.

6.1.1 Disorder Handling Components

Apart from the *SyncBuffer* and the K-Slacks applied on the two input streams introduced as in Section 4.1, our implementation consists of three additional components, the *late degree monitor* for each input stream, the *slack size adapter* and the *quality tracker*.

Late Degree Monitor

We apply a late degree monitor for each input stream. It maintains a vector R, to record the late degree of each incoming tuple within this stream. It is triggered whenever a tuple e comes, even before e goes into the K-Slack. The detailed algorithm is shown in Algorithm 3.

Algorithm 3: Late Degree Recording for each Input Stream

```
Input: Stream
Output: R
clk = 0;
R[*] = 0;
while e ← getIncomingTuple() do
    if e.ts > clk then
        clk ← e.ts;
    R[clk − e.ts] + +;                    ▷ record the late degree
```

Besides, intuitively, the older a record is, the less valuable it is. Therefore, we need to gradually eliminate the effect of old records in the late degree record vectors R_A and R_B for Stream A and B respectively. Algorithm 4 shows the pseudo code for this progress. The parameter $decay\_factor$ defines the speed the older record decays (see Section 6.1.2). We study the impact of parameter $decay\_factor$ in Section 6.3.2.3.

Algorithm 4: Late Degree Decaying

```
Input: R_A, R_B
Output: R_A, R_B
forall R_A[i] in R_A do
    R_A[i] ← R_A[i] * decay_factor;
forall R_B[i] in R_B do
    R_B[i] ← R_B[i] * decay_factor;
```

Quality Tracker

The quality tracker tracks the real quality of the JOIN operator, which is calculated by counting $Nr_{produced}$ and estimating $Nr_{original}$ with the data rates monitored by the ESP.

The quality tracker reports the quality of join results produced within a prespecified interval (denoted as $quality\_track\_duration$, see Section 6.1.2) at the end of each interval. We study the impact of parameter $quality\_track\_duration$ in Section 6.3.2.1.

Slack Size Adapter

The slack size adapter is triggered every $quality\_track\_duration$, after the quality tracker reports result quality. It executes Algorithm 2, to find a suitable pair of K_A and K_B for the next $quality\_track\_duration$.

After adjusting K_A and K_B, we execute Algorithm 4, to weaken the effect of old late degree records on new estimations.

6.1.2 Parameters

The proposed quality-driven disorder handler solution for sliding window join involves the following parameters, which can be configured by users.

quality_expectation	User's expectation on result quality, can be set to a decimal number between 0 and 1.
quality_track_duration	The time interval we estimate the result quality and do adaptation for.
optimize_overall_quality	If it is set to true, we perform slack size adaptation based on the overall result quality. By default, we perform slack size adaptation based on the result quality within each quality track duration.
k_granularity	The granularity of adjusting K.
decay_factor	The decay factor of late degree records, can be set to a decimal number between 0 and 1.

6.2 Experiment Setup

6.2.1 Hardware and Software Configuration

We use an HP Z620 workstation with an Intel® Xeon® CPU (24-cores @ 2.90GHz) and 96 GB RAM to implement and evaluate our proposal. The workstation runs SUSE 11.2 with Linux kernel 3.0.93. All codes are written in C++.

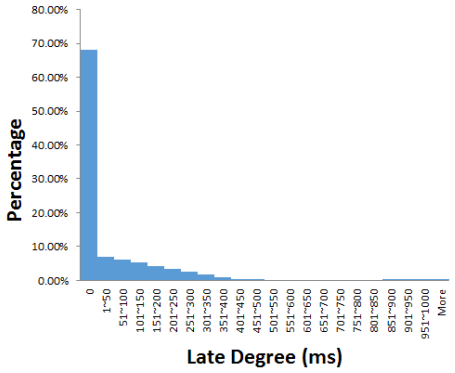
6.2.2 Datasets

We use two real-world datasets originating from a Realtime Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany [15]. Both datasets (denoted by `Soccer-16min` and `Soccer-23min`) have two out-of-order data streams, with are collected during soccer training games. Each tuple in the datasets contains a timestamp and a value. The time unit is microsecond. Table 6.1 shows the key statistics of each dataset.

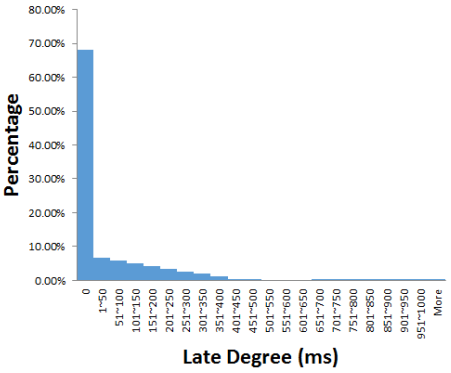
Dataset	Soccer-16min		Soccer-23min	
Stream	Stream A	Stream B	Stream A	Stream B
# of tuples	328597	324006	342690	429100
Total time(sec)	987.51	987.50	1436.88	1436.91

Table 6.1: Key statistics of Soccer-16min and Soccer-23min

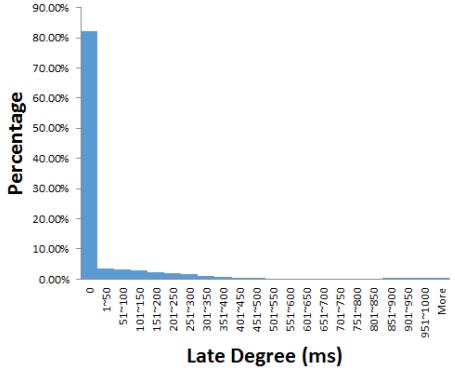
We use the overall late degree distribution of a stream to demonstrate its intra-stream disorder. Figure 6.1 shows the overall late degree distributions of the four input streams. As we can see, around 70% tuples are in-order, and most of the out-of-order tuples have a late degree of $\Lambda \in$ (0ms, 40ms].



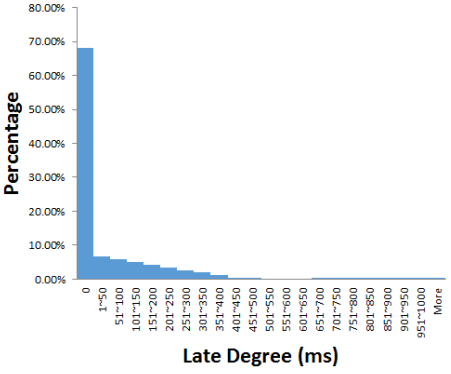
(a) Soccer-16min, Stream A



(c) Soccer-16min, Stream B



(b) Soccer-23min, Stream A



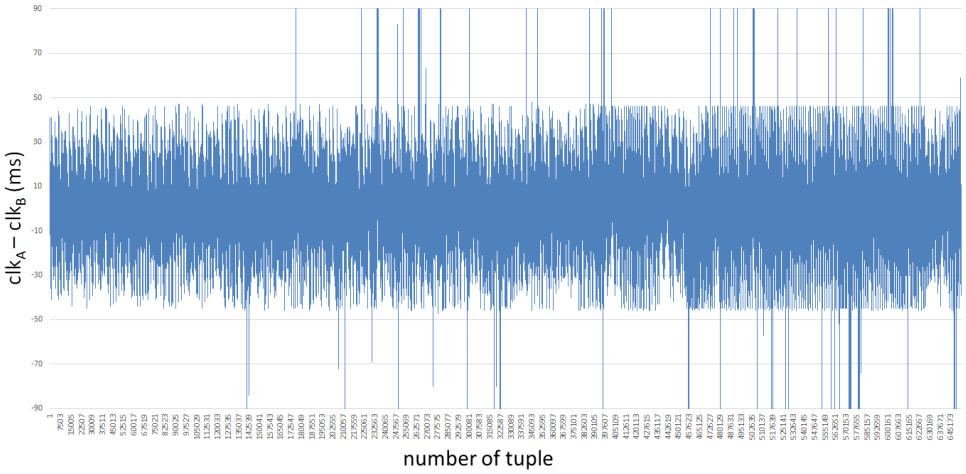
(d) Soccer-23min, Stream B

Figure 6.1: Overall late degree distribution of each stream used in the evaluation

In order to demonstrate the inter-stream disorder of the two datasets, we draw the out-of-sync spectrums for them in Figure 6.2. The x-axis is the number of tuple in the whole dataset, and the y-axis shows the difference between clk_A and clk_B at the time each tuple comes. As we can see, neither of the two streams stream is always leading or lagging, and the out-of-order degree vibrates within $\pm$50ms at most of the time.

6.2.3 Comparative Algorithm and Baselines

In order to evaluate the performance of our approach, we choose the Proportional-Derivative (PD) Controller [4] as a second slack size adaptation algorithm,



(a) Soccer-16min

(b) Soccer-23min

Figure 6.2: Out-of-sync spectrum of each dataset used in the evaluation

which is used in [22]. The PD controller calculates an *error* value as the difference between the result quality monitored by the quality tracker and the user-specified *quality_expectation*. Then it attempts to minimize the *error* by adjusting the K-Slack size. The parameters K_p and K_d are tuned manually, based on the Ziegler-Nichols method [23].

In addition, we use three baselines.

- `Baseline1`: no K-Slacks are applied;
- `Baseline2`: *SyncBuffer* and K-Slacks with minimum size that can guarantee 100% result quality are applied;
- `Baseline3`: no *SyncBuffer* nor K-Slacks are applied.

6.3 Experiment Results

In this section, we first use common parameter settings and compare the performance of our disorder handling approach with the PD controller and baselines in Section 6.3.1. Then we variate the parameters to see their influences to our approach in Section 6.3.2. At last, we measure the computational overhead caused by our disorder handling components in Section 6.3.3.

6.3.1 Basic Experiment

We use the following query to evaluate the algorithms. Window A has a size of two second, and Window B has a size of three seconds. The window slide unit U=10ms.

```
SELECT
  MAX(A.Timestamp, B.Timestamp) as Timestamp, A.Value, B.Value
FROM StreamA as A KEEP 2 SECOND, StreamB as B KEEP 3 SECOND
SLIDE 10 MILLISECOND
```

The parameter settings are listed in Table 6.2.

Parameter	Value
optimize_overall_quality	true/false
quality_track_duration	1000ms
k_granularity	10ms
decay_factor	0.8
quality_expectation	80%, 90%, 95%, 98%

Table 6.2: Standard parameter setup

In order to compare the performance of the algorithms, we measure two essential metrics for each query execution:

I. Quality Compliance Rate (QCR)

For each quality track duration, we test if the result quality within this duration meets the user-specified *quality_expectation*. The *quality compliance rate* is calculated as the number of quality track durations during which the actual result quality does not violate the user-specified *quality_expectation*, divided by the total number of quality track durations during the entire query execution.

II. Average Buffer Size (ABS)

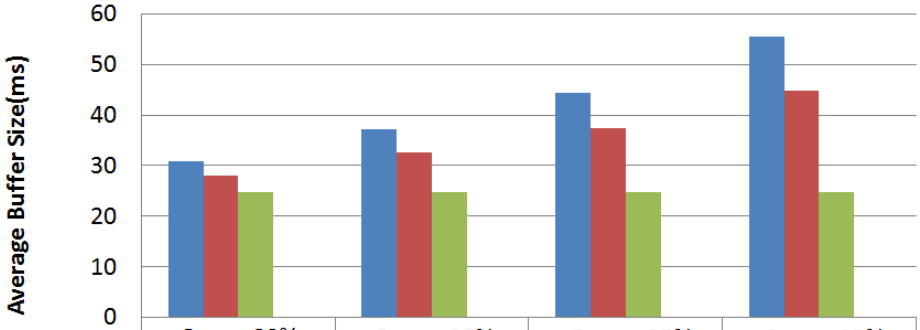
> The average buffer size within each quality track duration, including the size of K-Slacks and the size of *SyncBuffer*. It can be seen as the average result delay caused by the disorder handling.

We then calculate and compare QCR/ABS value. For the same dataset with coordinate setting, if Algorithm A produces higher QCR/ABS than Algorithm B, we consider A outperforms B, because it can trade one unit of delay time for higher result quality.

6.3.1.1 Buffer Size Adaptation based on Quality within Individual Quality Track Duration

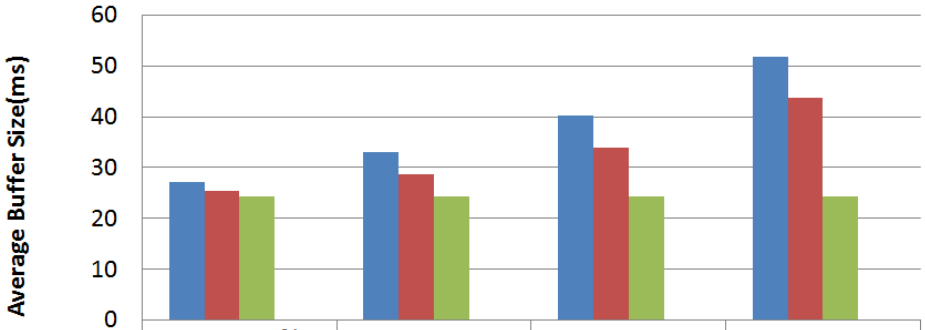
We first set *optimize_overall_quality* to false, namely, to perform buffer size adaptation based on the result quality within each individual quality track duration. We evaluate the performance of each algorithm and baseline over both `Soccer-16min` and `Soccer-23min`. The results are shown on the next pages.

The average buffer size (ABS) used in each execution is listed in Figure 6.3. Because `Baseline2` uses significantly large buffer, its buffer size is not shown in the bar chart. In general, the better the result quality that the user expects, the larger the buffer size that quality-driven buffer size adaptation approaches (our approach and the PD controller) use. Overall, our approach uses approximately 10% to 24% more buffer size compared with the PD controller, and the better the quality that the user expects, the larger the difference is.



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	30.75	37.2	44.31	55.53
PD	28.09	32.68	37.46	44.81
Baseline1	24.73	24.73	24.73	24.73
Baseline2	5490	5490	5490	5490
Baseline3	0	0	0	0

(a) Soccer-16min

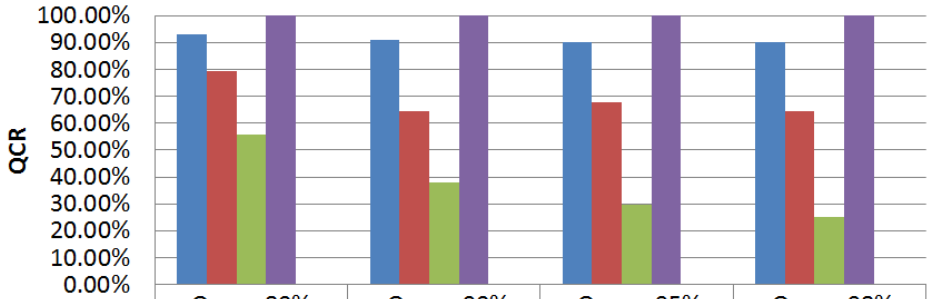


	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	27.06	32.93	40.13	51.72
PD	25.48	28.7	33.9	43.64
Baseline1	24.26	24.26	24.26	24.26
Baseline2	76910	76910	76910	76910
Baseline3	0	0	0	0

(b) Soccer-23min

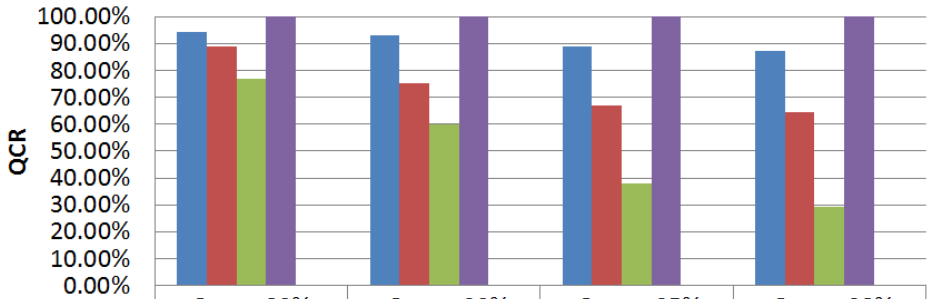
Figure 6.3: Average buffer size (ABS) ($optimize\_overall\_quality$ = false)

However, our approach has better quality compliance rate (QCR) performance than the PD controller, as shown in Figure 6.4. In general, though both quality-driven approaches exhibit decrease as the *quality_expectation* increases, our approach has higher QCR than the PD controller in every execution. Moreover, the PD controller has more serious decrease: from 79.21% to 64.40% for `Soccer-16min`, and from 88.99% to 64.32% for `Soccer-23min`; while the decrease of our approach is smaller: from 93.20% to 90.06% for `Soccer-16min`, and from 94.35% to 87.18% for `Soccer-23min`.



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	93.20%	91.18%	90.36%	90.06%
PD	79.21%	64.30%	67.65%	64.40%
Baseline1	55.88%	38.03%	29.51%	25.25%
Baseline2	100%	100%	100%	100%
Baseline3	0.20%	0%	0%	0%

(a) Soccer-16min

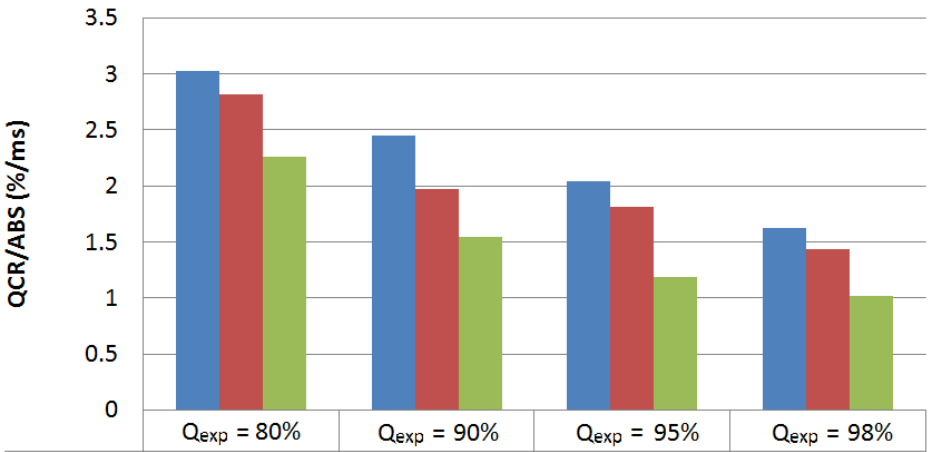


	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	94.35%	93.10%	89.13%	87.18%
PD	88.99%	75.40%	67.04%	64.32%
Baseline1	77.00%	59.93%	38.12%	29.41%
Baseline2	100%	100%	100%	100%
Baseline3	0.14%	0.14%	0.07%	0.07%

(b) Soccer-23min

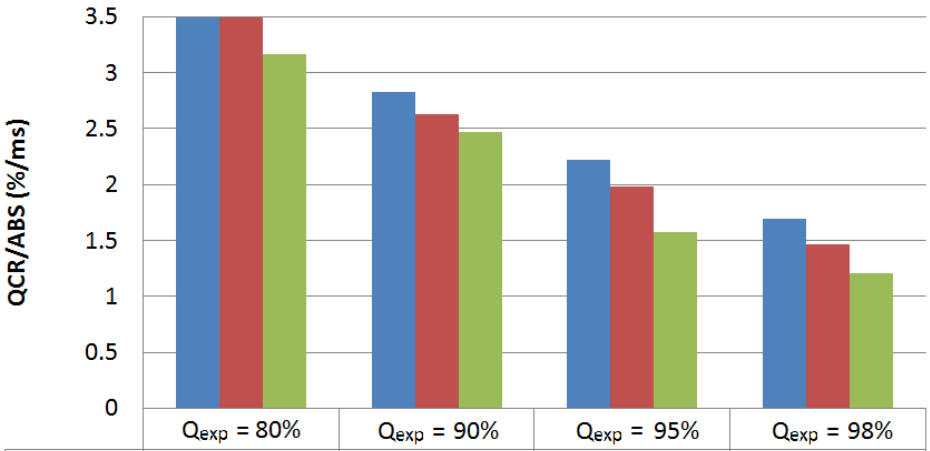
Figure 6.4: Quality compliance rate (QCR) ($optimize\_overall\_quality$ = false)

We further calculate the QCR/ABS value for each execution in Figure 6.5. From this figure we can see that, the QCR/ABS value decreases as the *quality_expectation* increases, which indicates that we need to trade more buffer size for higher result quality. Overall, our approach has better QCR/ABS performance than the PD controller, for both `Soccer-16min` and `Soccer-23min`, with the only exception when *quality_expectation* = 80% for `Soccer-23min`, where both of them has QCR/ABS=3.49.



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	3.03	2.45	2.04	1.62
PD	2.82	1.97	1.81	1.44
Baseline1	2.26	1.54	1.19	1.02

(a) Soccer-16min



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	3.49	2.83	2.22	1.69
PD	3.49	2.63	1.98	1.47
Baseline1	3.17	2.47	1.57	1.21

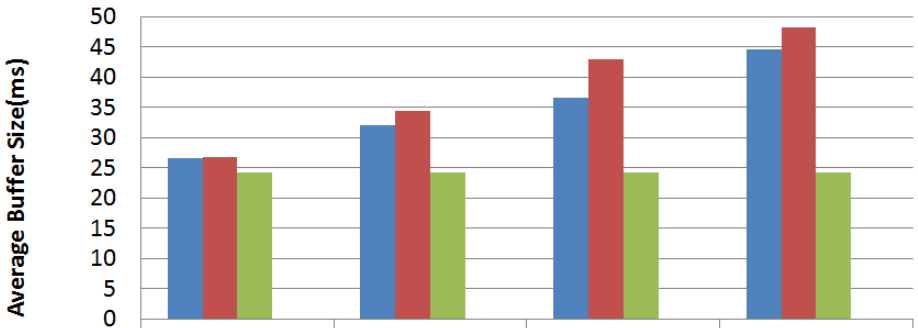
(b) Soccer-23min

Figure 6.5: QCR/ABS (*optimize_overall_quality* = false)

6.3.1.2 Buffer Size Adaptation based on Overall Quality

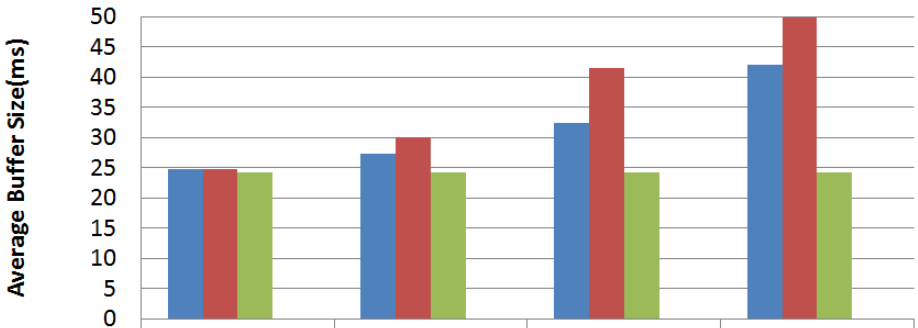
We define the overall quality as the result quality from the beginning of the execution to the current time, i.e., the total $Nr_{produced}$ divided by the total $Nr_{original}$ from the beginning of the execution. In case a user wants the overall quality to meet his requirement, he should set the parameter *optimize_overall_quality* to true. In this subsection, we evaluate the performance of each algorithm and baseline under this circumstance. The results are shown on the next pages.

As shown in Figure 6.6, similar to the case when we perform buffer size adaptation based on the result quality within individual quality track duration, the higher the result quality that the user requires, the bigger the buffer size that we need. However, when we perform buffer size adaptation based on overall quality, our approach uses less buffer size on average than the PD controller in most cases, which is opposite to the situation when based on the result quality within individual quality track duration.



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	26.64	32.03	36.57	44.68
PD	26.77	34.38	42.98	48.25
Baseline1	24.73	24.73	24.73	24.73
Baseline2	5490	5490	5490	5490
Baseline3	0	0	0	0

(a) Soccer-16min

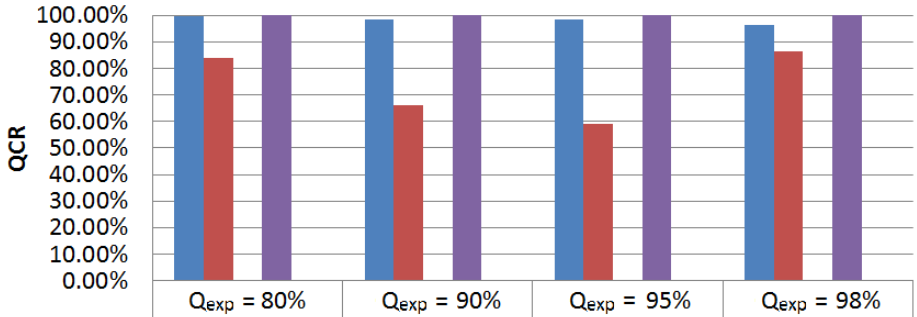


	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	24.78	27.26	32.38	42.11
PD	24.75	30.01	41.54	49.93
Baseline1	24.26	24.26	24.26	24.26
Baseline2	76910	76910	76910	76910
Baseline3	0	0	0	0

(b) Soccer-23min

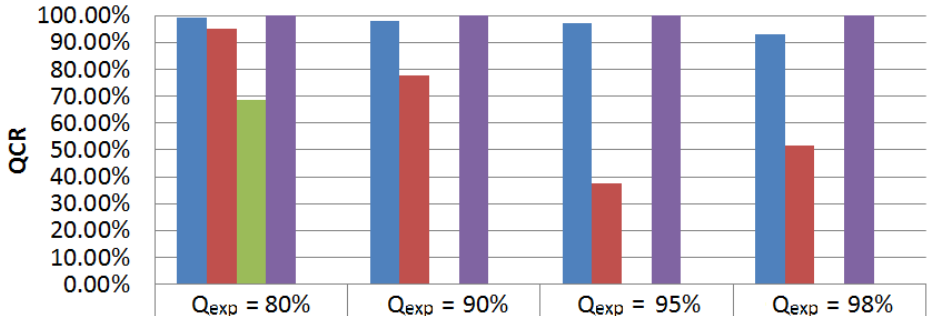
Figure 6.6: Average buffer size (ABS) ($optimize\_overall\_quality$ = true)

From Figure 6.7 we can see that, QCR decreases as the *quality_expectation* increases in nearly all cases, except for the PD controller when *quality_expectation* is set to 98% in both datasets. `Baseline1` has the most significant decrease. Its overall quality seldom meets the user's expectation, when the user has high quality requirement. However, QCR of our approach does not demonstrate significant decrease as the *quality_expectation* increases, compared to the PD controller. With the worst case when *quality_expectation* = 98% in `Soccer-23min`, our approach can still get a QCR of 93.17%. The final result quality of `Soccer-23min` when *quality_expectation* = 98% with the PD controller is 97.88%, not fulfilling user's demand.



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	99.80%	98.56%	98.68%	96.35%
PD	83.77%	66.23%	59.13%	86.31%
Baseline1	0.00%	0.00%	0.00%	0.00%
Baseline2	100%	100%	100%	100%
Baseline3	0.00%	0.00%	0.00%	0.00%

(a) Soccer-16min

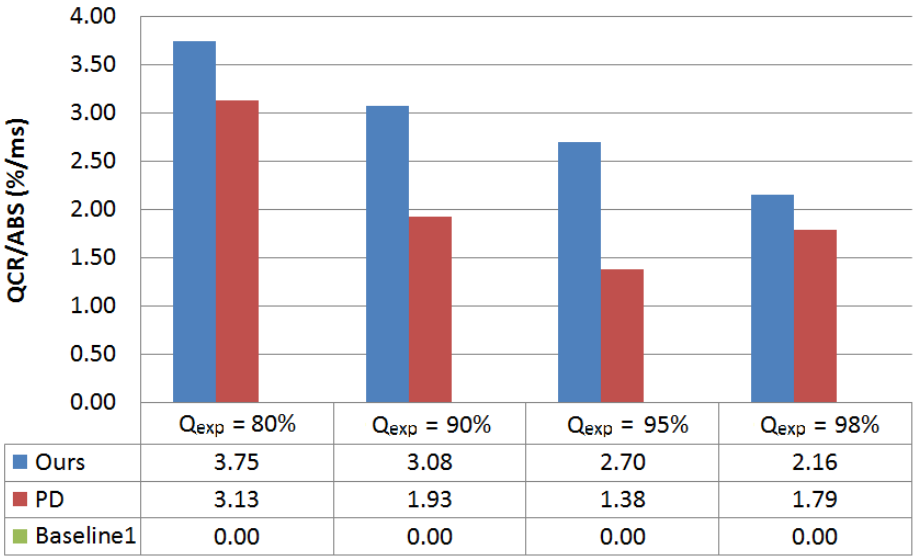


	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	99.44%	98.11%	97.21%	93.17%
PD	95.19%	77.70%	37.35%	51.78%
Baseline1	68.57%	0.14%	0.00%	0.00%
Baseline2	100%	100%	100%	100%
Baseline3	0.00%	0.00%	0.00%	0.00%

(b) Soccer-23min

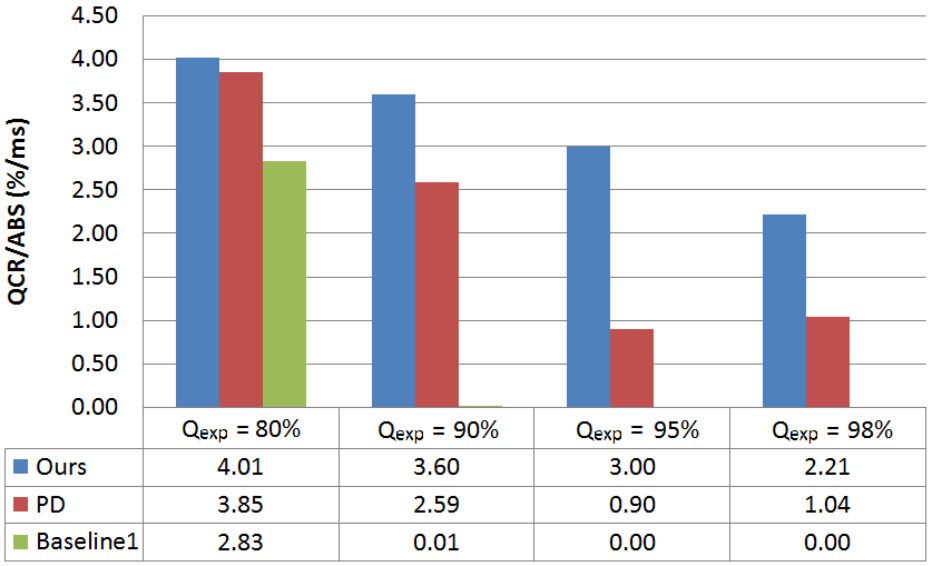
Figure 6.7: Quality compliance rate (QCR) (*optimize_overall_quality* = true)

Since in most cases our approach has smaller ABS and higher QCR than the PD controller, it is obvious that our approach would have higher QCR/ABS compared to the PD controller, as corroborated by the results in Figure 6.8. Overall, when we perform buffer size adaptation based on overall quality, our approach also has higher QCR/ABS than the PD controller, for both `Soccer-16min` and `Soccer-23min`.



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	3.75	3.08	2.70	2.16
PD	3.13	1.93	1.38	1.79
Baseline1	0.00	0.00	0.00	0.00

(a) Soccer-16min



	Q_{exp} = 80%	Q_{exp} = 90%	Q_{exp} = 95%	Q_{exp} = 98%
Ours	4.01	3.60	3.00	2.21
PD	3.85	2.59	0.90	1.04
Baseline1	2.83	0.01	0.00	0.00

(b) Soccer-23min

Figure 6.8: QCR/ABS ($optimize\_overall\_quality$ = true)

6.3.1.3 Comparison between the Two Quality Targets

In order to compare the two result quality targets (result quality within individual quality track duration and the overall quality) based on which we perform buffer size adaptation, we cross compare the experiment results in Section 6.3.1.1 and Section 6.3.1.2.

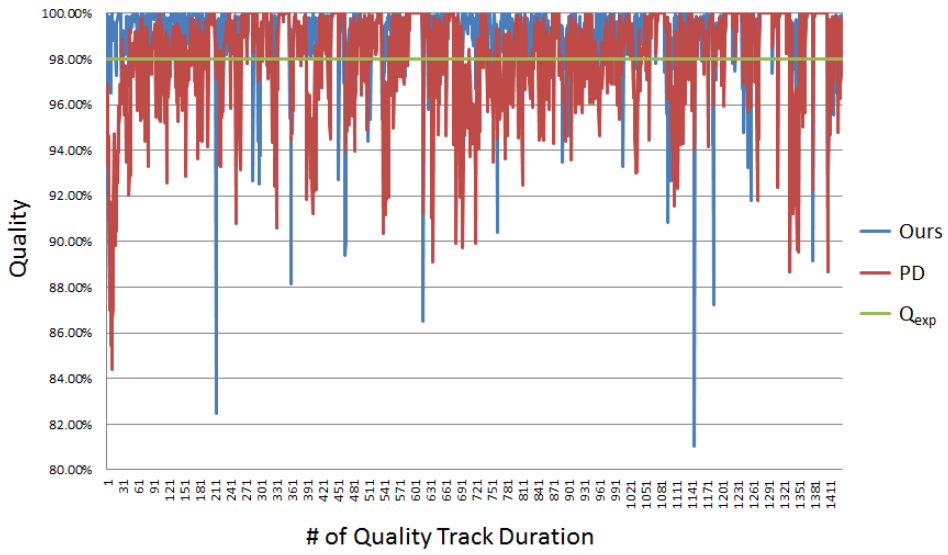
Cross comparing Figure 6.3 and Figure 6.6 we can see that, for both datasets and all *quality_expectation* settings, our approach consumes less average buffer size when we perform buffer size adaptation based on overall quality, than based on result quality within individual quality track duration. The PD controller has the opposite situation, except for the case where *quality_expectation* is set to 80%, the PD controller also consumes less average buffer size when performing buffer size adaptation based on overall quality, than based on result quality within individual quality track duration.

Cross comparing Figure 6.4 and Figure 6.7 we can see that, for both datasets and all *quality_expectation* settings, our approach achieves better QCR when performing buffer size adaptation based on overall quality, than based on result quality within individual quality track duration. We do not see a clear pattern for the PD controller.

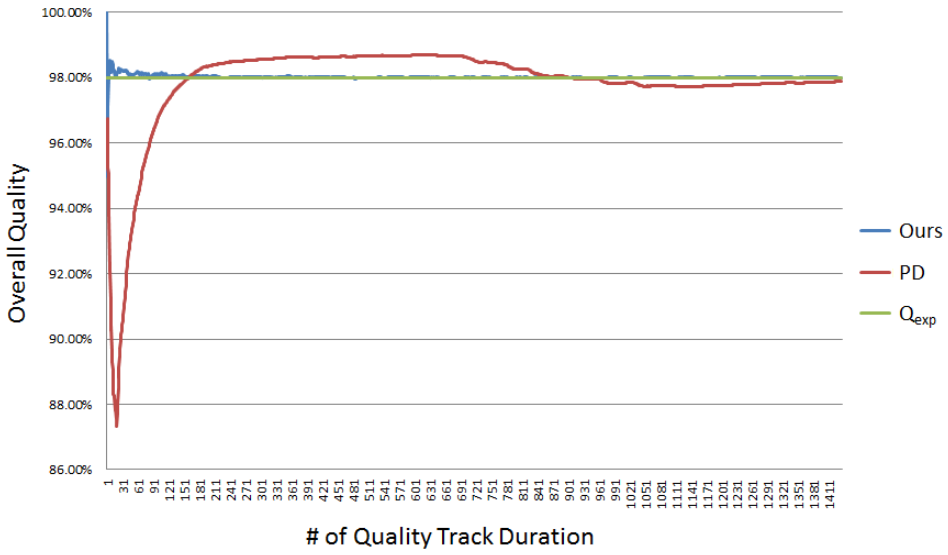
Figure 6.9 presents the result quality change during the whole executions using common settings, one performing buffer size adaptation based on result quality within individual quality track duration (Figure 6.9a), and the other based on overall quality (Figure 6.9b). The curves of both our approach and the PD controller in Figure 6.9b are much smoother than the ones in Figure 6.9a. The reason is that, the result quality of one quality track duration has little impact on the result quality of another; while as the total $Nr_{produced}$ and $Nr_{original}$ grow bigger, the overall quality calculated in one quality track duration has higher and higher impact on the overall quality calculated in the next quality track duration.

This also explains why the the PD controller has bad performance when performing buffer size adaptation based on overall quality. According to the algorithm of the PD controller, it only increases K when the overall quality is smaller than *quality_expectation*. However, when performing buffer size adaptation based on overall quality, in the late period of the execution, when the the total $Nr_{produced}$ and $Nr_{original}$ is very large, even an overall quality which is slightly smaller than *quality_expectation* has significant impact on the later values of overall quality. However, according to the algorithm of the PD controller, it will only increase K a little bit, which cannot stop the further decrease of the overall quality. This case is shown in Figure 6.9b, the overall quality of the PD controller continues to decrease after it gets smaller than *quality_expectation*. On the contrary, our approach adjusts K based on estimation of future result quality. Therefore, it could increase K even if the

overall quality is not smaller than *quality_expectation*.



(a) Optimizing quality for each quality track duration



(b) Optimizing overall quality

Figure 6.9: Quality plot (Soccer-23min, *quality_expectation*=98%)

Cross comparing Figure 6.5 and Figure 6.8, we can see that, our approach gets higher QCR/ABS in most cases. This indicates that our approach is more suitable than the PD controller for quality-driven buffer size adaptation, since it can trade oneunit of delay time for higher result quality. Besides, the QCR/ABS of our approach is higher in Figure 6.8 than in Figure 6.5, which indicates that it is easier for our approach to provide better result quality

with smaller buffer size (i.e., smaller result delay), when it performs buffer size adaptation based on overall quality, than based on result quality within individual quality track duration.

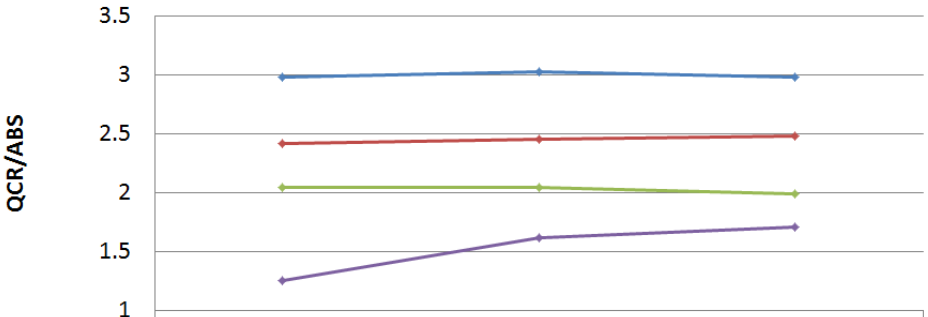
6.3.2 Influence of Parameters

In this subsection, we evaluate the influence of the three parameters: *quality_track_duration*, $k\_granularity$ and $decay\_factor$ to our approach. For each of them, we fix the other parameters and test the QCR/ABS performance under varying settings of this parameter, with both datasets and the four *quality_expectation* settings: 80%, 90%, 95% and 98%. We take the default setting of quality target, i.e., to perform buffer size adaptation based on result quality within individual quality track duration.

6.3.2.1 Influence of *quality_track_duration*

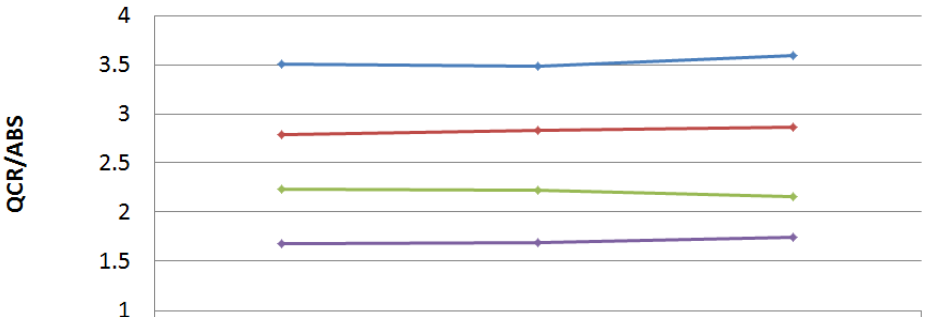
As we have already explained in Section 5.3.3, the setting of the time interval T (i.e., parameter *quality_track_duration*) for which we estimate the result quality is important to our approach. In order to evaluate its influence, we fix other parameters and vary *quality_track_duration* from 500ms to 2000ms, and compare their QCR/ABS performance.

From Figure 6.10 we can see that, for both `Soccer-16min` and `Soccer-23min`, the QCR/ABS performance of our approach under three different settings of *quality_track_duration* are roughly equivalent (with a deviation of ±3% compared to the standard setting 1000ms), when *quality_expectation* is set to 80%, 90% and 95%. When we have *quality_expectation*=98%, for `Soccer-16min`, the QCR/ABS performance under the setting *quality_track_duration* = 500ms is significantly decreased (-22.8% compared to the standard setting); while the QCR/ABS performance under the setting *quality_track_duration* = 2000ms is seen improved by 5.6% compared to the standard setting. The reason could be that, `Soccer-16min` has fewer result tuples per second. In this case, according to the discussions in Section 5.3.3, when we have shorter *quality_track_duration*, the result quality estimation is less accurate, vice versa.



	quality_track_duration		
	500ms	1000ms	2000ms
Q_{exp}=80%	2.98	3.03	2.98
Q_{exp}=90%	2.42	2.45	2.48
Q_{exp}=95%	2.04	2.04	1.99
Q_{exp}=98%	1.25	1.62	1.71

(a) Soccer-16min



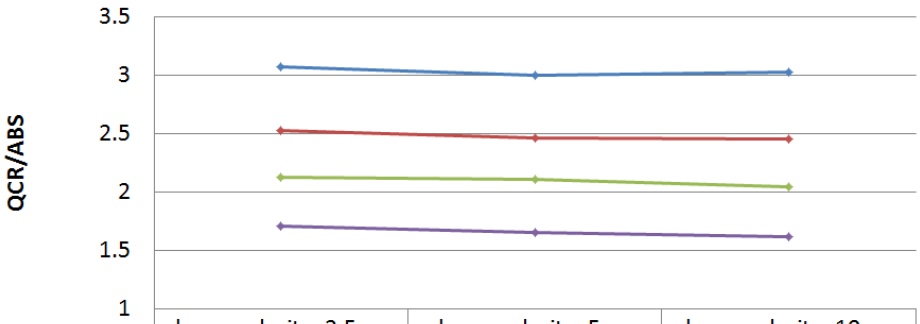
	quality_track_duration		
	500ms	1000ms	2000ms
Q_{exp}=80%	3.51	3.49	3.59
Q_{exp}=90%	2.79	2.83	2.86
Q_{exp}=95%	2.23	2.22	2.16
Q_{exp}=98%	1.67	1.69	1.74

(b) Soccer-23min

Figure 6.10: Influence of $quality\_track\_duration$ ($k\_granularity$ = 10ms, $decay\_factor$ = 0.8)

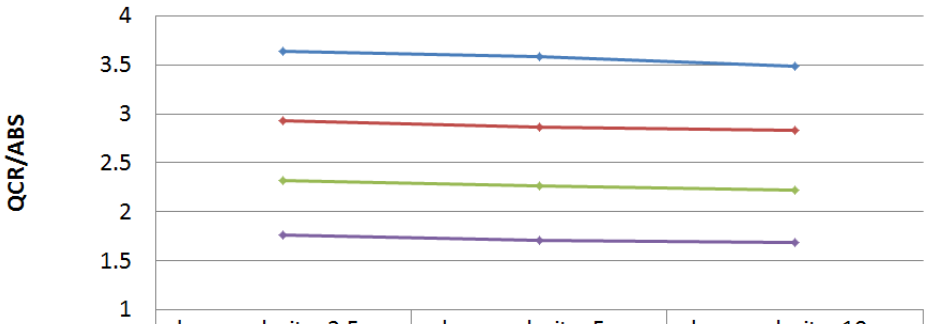
6.3.2.2 Influence of *k_granularity*

The parameter *k_granularity* defines the granularity we adjust the size of *K*-Slacks. We fix other parameters and vary *k_granularity*, in order to find its influence on the performance of our approach. Figure 6.11 shows the QCR/ABS performance when *k_granularity* is set to 2.5ms, 5ms and 10ms.



	k_granularity=2.5ms	k_granularity=5ms	k_granularity=10ms
Q_{exp}=80%	3.07	3.00	3.03
Q_{exp}=90%	2.53	2.46	2.45
Q_{exp}=95%	2.13	2.11	2.04
Q_{exp}=98%	1.71	1.65	1.62

(a) Soccer-16min



	k_granularity=2.5ms	k_granularity=5ms	k_granularity=10ms
Q_{exp}=80%	3.64	3.58	3.49
Q_{exp}=90%	2.93	2.86	2.83
Q_{exp}=95%	2.32	2.26	2.22
Q_{exp}=98%	1.76	1.71	1.69

(b) Soccer-23min

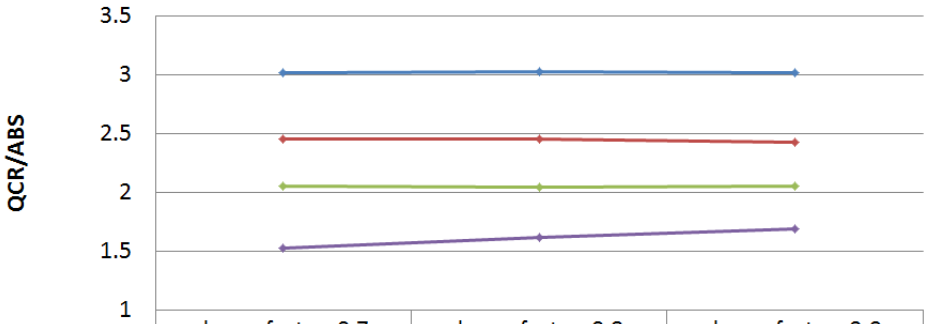
Figure 6.11: Influence of *k_granularity*
(*quality_track_duration* = 1000ms, *decay_factor* = 0.8)

As shown in this figure, our approach has better performance with smaller *k_granularity* settings than with larger *k_granularity* settings in most cases.

This can be explained by how we determines the buffer size. According to Algorithm 2, we start the incremental search for suitable K values from value zero, until the estimated quality is not smaller than *quality_expectation*. Therefore, a smaller *k_granularity* setting will have a chance to result in smaller K values, hence smaller buffer size. However, due to estimation errors, this smaller buffer size can sometimes result in a result quality lower than *quality_expectation*. This explains the counterexample in `Soccer-16min`, when *quality_expectation* is 80%; the QCR/ABS of the setting *k_granularity* = 5ms slightly worse than the QCR/ABS of the setting *k_granularity*=10ms.

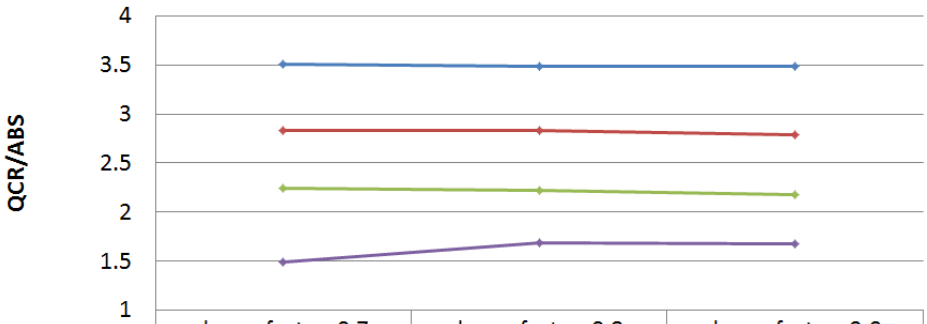
6.3.2.3 Influence of *decay_factor*

The parameter $decay\_factor$ controls how fast the system eliminates the influence of old late records, as explained in Section 6.1.2 (the smaller the faster). In order to analyze the influence of $decay\_factor$ to our approach, we fix other parameters and vary $decay\_factor$. Figure 6.12 shows the QCR/ABS performance when $decay\_factor$ is set to 0.7, 0.8 and 0.9.



	decay_factor=0.7	decay_factor=0.8	decay_factor=0.9
Q_{exp}=80%	3.02	3.03	3.02
Q_{exp}=90%	2.45	2.45	2.43
Q_{exp}=95%	2.05	2.04	2.05
Q_{exp}=98%	1.53	1.62	1.69

(a) Soccer-16min



	decay_factor=0.7	decay_factor=0.8	decay_factor=0.9
Q_{exp}=80%	3.51	3.49	3.48
Q_{exp}=90%	2.83	2.83	2.79
Q_{exp}=95%	2.24	2.22	2.18
Q_{exp}=98%	1.49	1.69	1.67

(b) Soccer-23min

Figure 6.12: Influence of $decay\_factor$
($quality\_track\_duration$ = 1000ms, $k\_granularity$ = 10ms)

From the figure we can see that, when $quality\_expectation$ is set to 80%, 90% and 95%, there is little influence of the $decay\_factor$ setting (with a de-

viation of ±2% compared to the standard setting 0.8). Only when we set *quality_expectation* to 98%, the setting $decay\_factor = 0.7$ has significant decrease of performance (-5.6% for `Soccerr-16min`, -11.8% for `Soccerr-23min`, compared to the standard setting). This indicates that, it is not always better to eliminate the influence of old late records fast, because sometimes they are still valuable in revealing the pattern of tuples. Besides, it is also not optimal to do very slow decay, since we always want the newest records to have the biggest influence. Overall, our approach has the best performance when $decay\_factor$ is set to 0.8 in our test.

6.3.3 Computational Overhead

In order to evaluate the computational overhead of the three disorder handling components, the *late degree monitor*, the *slack size adapter* and the *quality tracker*, as mentioned in Section 6.1.1, we measure and compare the execution times accomplishing JOIN executions over the same dataset with and without them.

Technically, we set a flag to enable and disable the three components in the code and use the same sorted data stream as input for these two different executions. The reason why the original unsorted data stream is not used is that, if we disable the three components, the inter-stream disorder will not be handled the same way as we enable them. This will lead to different join results because the operands of the join operator will be different. More specifically, when we disable the three components, more input tuples will be discarded, and, as a consequence, fewer join results will be produced, than when we enable the three components. Since join operations also take time, different join operations make it difficult to evaluate the execution time difference caused by the three disorder handling components. Using sorted data stream will totally avoid this problem, because the join operands will keep the same between the entire execution with disorder handling components, and the entire execution without them. In our experiment we use sorted `Soccer-23min` as input. We measure the total execution time five times for each condition and calculate the average total execution time. The results are listed in Table 6.3.

No. experiment	Enable components	Disable components
1	1,375.440	1,364.174
2	1,366.799	1,386.612
3	1,379.475	1,346.368
4	1,346.852	1,367.247
5	1,377.083	1,382.139
Average	1,369.130	1,369.308
Standard error	13.340	15.976

Table 6.3: Execution time comparison (unit:sec)

The average execution time shows that the computational overhead caused by the three disorder handling components is negligible.

Chapter 7

Conclusion

In this thesis, we have focused on quality-driven sliding-window JOIN over data streams. Based on existing works, we have proposed our own a probabilistic model, aiming to reveal the relationship between buffer size and result quality. With estimation of future result quality based on observed late arrival pattern of the incoming data, our model can minimize the result latency while fulfilling users' requirement on result quality. We have implemented our proposal in SAP Sybase ESP system.

In the evaluation part, we have validated our quality-driven slack size adaptation approach using real world data. Experimental results have shown that, compared to the existing PD controller algorithm, our approach is more suitable for quality-driven slack size adaptation, when performing sliding-window JOIN over data streams. The influence of parameters to our approach has also been evaluated.

Although the implemented solution has shown its effectiveness, there exists a wide range of aspects for future works and improvements. First, we have only implemented sliding-window JOIN over two data streams. In principle, our approach can be extended and used in a variety of operators (e.g., UNION, INTERSECT) over multiple data streams. Second, in the evaluation part, we have only tested the case when the join selectivity $\rho = 100\%$, while the case when ρ is not always 100% remains unstudied.

List of Notations

λ_r	Join result latency
Λ	Late degree of a tuple
ρ	Join selectivity
a	Input tuples in Stream A
b	Input tuples in Stream B
D_S	Late degree distribution of Stream S
$E(Nr_{produced})$	The expected number of actually produced result tuples
$E(Q)$	Expected result quality
$e.ts_{process}$	The largest timestamp that has been processed when e arrives
K	Size of a K-Slack
L	Size of *SyncBuffer*
$Nr_{original}$	The number of result tuples that should be produced over a period of time, if there is no disorder in input streams
$Nr_{original}(t)$	The number of original result tuples for the time unit t
$Nr_{produced}$	The number of result tuples that are actually produced over a period of time
$Nr_{produced}(t)$	The number of result tuples produced for the time unit t
Q	Result quality
Q_{est}	Estimated result quality
Q_{exp}	The quality expectation of the user
r	Result tuples in Stream R
$StreamS\prime$	The output stream of K-Slack
U	Window slide unit

W	The size of a window
$WindowEnd$	Current common endpoint of both windows
$SyncBufA$	Synchronization buffer of Stream A
$SyncBufB$	Synchronization buffer of Stream B

List of Figures

List of Tables

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] 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. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[3] Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 209–218, New York, NY, USA, 2000. ACM.

[4] Karl Johan Aström and Tore Hägglund. Pid controllers: theory, design and tuning. 1995.

[5] 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*, pages 419–430. ACM, 2004.

[6] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, September 2004.

[7] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[8] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115*, 2006.

[9] 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, pages 89–98, New York, NY, USA, 2008. ACM.

[10] Moustafa A. Hammad. Optimizing in-order execution of continuous queries over streamed sensor data. In *In Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM*, pages 143–146, 2005.

[11] 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*, pages 1081–1092. ACM, 2010.

[12] 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*, pages 311–322. ACM, 2005.

[13] Christopher Mutschler and Michael Philippsen. Distributed low-latency out-of-order event processing for high data rate sensor streams. *Parallel and Distributed Processing Symposium, International*, 0:1133–1144, 2013.

[14] 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, pages 147–158, New York, NY, USA, 2013. ACM.

[15] 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*, pages 289–294. ACM, 2013.

[16] Kostas Patroumpas and Timos Sellis. Window specification over data streams. In *Current Trends in Database Technology–EDBT 2006*, pages 445–464. Springer, 2006.

[17] P. Révész. The laws of large numbers. Probability and Mathematical Statistics. A Series of Monographs and Textbooks. 4. New York-London: Academic Press. 176 p. (1968)., 1968.

[18] SAP Sybase ESP. `http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html`.

[19] 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, pages 263–274, New York, NY, USA, 2004. ACM.

[20] Peter A Tucker. *Punctuated data streams*. PhD thesis, Oregon Health & Science University, 2005.

[21] Ji Wu, K-L Tan, and Yongluan Zhou. Window-oblivious join: a data-driven memory management scheme for stream join. In *Scientific and Statistical Database Management, 2007. SSBDM'07. 19th International Conference on*, pages 21–21. IEEE, 2007.

[22] Hongjin Zhou. Quality-driven out-of-order handling for window aggregates over data streams. Master Thesis, July 2014.

[23] John G Ziegler and Nathaniel B Nichols. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, June 8, 2015