10-2008

Window Queries Over Data Streams

Jin Li
Portland State University

ABSTRACT

An abstract of the dissertation of Jin Li for the Doctor of Philosophy in Computer Science presented October 17, 2008.

Title: Window Queries over Data Streams

Evaluating queries over data streams has become an appealing way to support various stream-processing applications. Window queries are commonly used in many stream applications. In a window query, certain query operators, especially blocking operators and stateful operators, appear in their windowed versions. Previous research work in evaluating window queries typically requires ordered streams and this order requirement limits the implementations of window operators and also carries performance penalties. This thesis presents efficient and flexible algorithms for evaluating window queries. We first present a new data model for streams, *progressing streams*, that separates stream progress from physical-arrival order. Then, we present our window semantic definitions for the most commonly used window operators—window aggregation and window join. Unlike previous research that often requires ordered streams when describing window semantics, our window semantic definitions do not rely on physical-stream arrival properties. Based on the window semantic definitions, we present new implementations of window aggregation and

window join, *WID* and *OA-Join*. Compared to the existing implementations of stream query operators, our implementations do not require special stream-arrival properties, particularly stream order. In addition, for window aggregation, we present two other implementations extended from WID, *Paned-WID* and *AdaptWID*, to improve excution time by sharing sub-aggregates and to improve memory usage for input with data distribution skew, respectively. Leveraging our order-insenstive implementations of window operators, we present a new architecture for stream systems, *OOP* (*O*ut-of-*O*rder *P*rocessing). Instead of relying on ordered streams to indicate stream progress, OOP explicitly communicates stream progress to query operators, and thus is more flexible than the previous in-order processing (IOP) approach, which requires maintaining stream order. We implemented our order-insensitive window query operators and the OOP architecture in NiagaraST and Gigascope. Our performance study in both systems confirms the benefits of our window operator implementations and the OOP architecture compared to the commonly used approaches in terms of memory usage, execution time and latency.

WINDOW QUERIES OVER DATA STREAMS

by

JIN LI

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University

Acknowledgments

I am deeply grateful to David Maier, my thesis adviser, for his support and guidance. I am very fortunate to have Dave as my adviser. His broad knowledge amazes me and spoils his students (including me). Instead of doing related-work research, we often "ask Dave". Dave has the most brilliant research ideas. He also has the amazing ability and patience to help me sort out initial, random research ideas. He is a role model I can learn from in many aspects. I respect his dedication to education and research and his generosity and patience in helping others.

I am in debt to Kristin Tufte for her continual help and encouragement along my Ph.D. journey. She worked with me side-by-side to improve my writing skills and presentation skills, and to help me refine research ideas. Kristin is also a true friend. Her support and encouragement is crucial for me to finish my thesis.

I would like to thank the rest of my thesis committee members, Theodore Johnson, Leonard Shapiro, Wu-chi Feng, and Robert L. Bertini, for various discussions and suggestions. I also appreciate the whole database group at OGI and later at PSU for the very active and supportive study environment.

I thank my family's understanding and support. I especially thank my husband, Sun Yi, for his patience, support, and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

The input data of many modern applications naturally takes the form of data streams (instead of static stored data sets), such as network packets, web-click streams, environmental-sensor data, phone-call records, online auctions and bids, cheat detection in computer games, and stock quotes. Many stream-monitoring applications need to process high-volume streams while providing low-latency responses, and thus require on-the-fly processing. For example, both computer networks and financial markets may generate hundreds of thousands of data items per minute, and the monitoring systems must provide real-time information so that their clients (e.g., network-traffic diagnosis systems and traders) can make the correct decisions regarding the current situation. Just as traditional database queries serve as an easy-to-use, declarative, scalable way to process stored relational data, so to have stream queries, which are similar queries over data streams, has been gradually adopted as a beneficial approach for online stream processing. Several stream-query engines have been built in the research domain [2, 4, 12, 46, 63] and a few have been put to use for real-world applications. For example, Gigacope is a stream-query system that specifically supports network-traffic monitoring [14]; StreamBase [65] and Truviso [70] are both more general stream-query engines that support various applications, such as financial services, telecommunication monitoring, and military systems.

Stream queries differ from relational-database queries in two main ways. First, users of stream-query systems are often more interested in querying recent data in the stream and having the query results updated periodically than in getting the information over the entire history of the stream. (Note that standard relational query evaluation techniques only support one-time evaluation, rather than periodic updates.) Second, as data streams are potentially unbounded, a blocking operator (e.g., aggregation), which require the entire input data set before producing any results, may have its output delayed indefinitely; and a stateful operator (e.g., join) may need to maintain an unbounded amount of state. Therefore, stream systems often make restrictions on the types of blocking and stateful operators allowed to ensure that queries can be unblocked and the state that they need to maintain does not grow without bound. For example, stream systems may allow only aggregations that can potentially be unblocked. This condition translates to the requirement that each group in an aggregation must eventually be complete, even though the input stream is unbounded. The requirement indicates that aggregations in stream queries usually need to have a grouping condition on a special ordering attribute (e.g., a timestamp attribute), and thus these queries can output results as the timestamp value increases. (In some special cases groups end naturally; for example, when each group is a different auction in an online auction system.) Similarly, stream systems require a joining condition that ensures that the join can purge state. For example, an equality-join predicate on the timestamp of the input streams can support join-state purging.

A window is a special condition specified for stream query operators. It is often defined on an ordering attribute and is very commonly used in stream queries with blocking and stateful operators. Window aggregation is an aggregation with a special grouping condition on the ordering attribute that maps each tuple to one or more groups. For join, a window is used to limit the range of tuples in one input with which each tuple of the other input may join and thus it limits the amount of state that the operator needs to maintain. In query Q1-1 shown below, a network-traffic-monitoring system can use a windowed aggregate operator to count the number of packets from each source IP in a link, M, for the past 10 minutes, advancing at 1-minute intervals. We assume the (simplified) schema of the packets in M is <*srcIP*, *srcPort*, *destIP*, *destPort*, *len*, *ts*>. Here, *srcIP* and *srcPort* are the source IP and port of the packet, respectively; *destIP* and *destPort* are the destination IP and port of the packet, respectively; *len* is the size of the packet and *ts* is the timestamp.

Q1-1: "Count the number of packets from each source IP for the past 10 minutes; update the results every minute."

```
SELECT    srcIP, count(*) [RANGE 10 minutes, SLIDE 1 minute, WA ts]
FROM      M
GROUP BY  srcIP
```

Here, RANGE, SLIDE, and WA are called window parameters. These parameters collectively specify a "window of interest" that separates the input stream into potentially overlapping sub-streams, which we call *window extents*. RANGE is the size of the window, SLIDE is the distance that the window moves each time it advances, and WA is the *windowing attribute* on which RANGE and SLIDE are

defined. For Q1-1, window extents are overlapping 10-minute sub-streams, for example, [00:10:00, 00:20:00), [00:11:00, 00:21:00), [00:12:00, 00:22:00), defined on the *ts* attribute. These window extents can be viewed as special groups whose member tuples may also belong to other groups. One way to view window aggregation is as a seperate aggregate being computed for each window extent.

We briefly review the current commonly used approach for evaluating Q1-1, and then discuss the issues with each approach. (We will expand this analysis in Chapter 6.) The existing approach, which we term the *buffering* technique, assumes that the input stream M is ordered, and maintains input tuples in a buffer until they no longer belong to the current window extent. It determines window boundaries based on the requirement that tuples are ordered. That is, the arrival of the first tuple outside of a window extent closes the extent. When a window extent ends, the buffering technique computes the aggregate over the buffered tuples, and then purges expired tuples from the buffer. We see some obvious issues with the buffering technique. First, it requires a tuple buffer to materialize each window extent. Second, the content of each window extent tends to be tied to window operator implementations and physical stream properties such as stream-arrival order. It requires ordered input streams; as we will discuss later, out-of-order arrival of tuples arises natually in stream-processing systems due to causes such as variation in transmission delays. If data is not in order, a sorting mechanism such as Aurora's BSort [4] must be used to reorder the data. Enforcing order incurs performance overhead such as memory and latency, and also constrains the implementation of windowed-query evaluation. Third, the buffering

technique relies on the physical arrival of tuples to determine window boundaries, and thus stream abnormalities such as lulls (i.e., periods of non-arrival of tuples) cause troubles: Lulls in physical tuple arrival may delay result generation, and thus special mechanisms such as time-outs have to be introduced for handling lulls.

We believe that a root cause of these issues with this current approach is a lack of logical[1] definitions for stream query operators. Logical definitions of query operators, independent of the physical properties of data and data storage, are one of the most important benefits of relational database systems. Logical definitions of query operators allow *logical independence* of queries: Users can focus on the meaning of their queries, regardless of physical data properties; one query operator may have alternative physical implementations that optimize for different physical properties, from which the query system can choose. Logical independence is also important for stream-query operators. Previously, the semantics of window operators were often discussed "operationally" and assumed ordered and continuous input streams, and thus these "operational" semantics led to order-sensitive and often inefficient implementations of operators. More importantly, these order-sensitive implementations of window operators lead to a commonly used stream-query evaluation architecture, which we term *IOP* (*I*n-*O*rder *P*rocessing). The IOP architecture assumes that streams in a stream system are ordered, and thus implementations of query operators can rely on the order to output results or purge

[1] Here, "logical" refers to semantics that is independent of "physical" implementation details. Logical window semantics defines the content of each window extent; or, equivalently, it defines the window membership of each input tuple.

state. Stream systems using the IOP architecture enforce order for input streams and require that all stream-query operators maintain stream order in their output. However, we find this approach is often inflexible and inefficient, especially in terms of memory and latency.

The focus of this thesis is efficient evaluation of window stream queries, which includes a new stream model and semantic definitions of window stream-query operators, order-agnostic implementations of the operators motivated by the new semantic definitions, and also an alternative stream-system architecture, which we term *OOP* (*O*ut-of-*O*rder *P*rocessing), in contrast to IOP. The OOP architecture is a natural extension of the order-agnostic implementation of stream query operators. In the OOP architecture, streams carry explicit progress information (e.g., punctuation, which are special tuples indicating ends of subsets of regular tuples). Implementations of query operators rely on that progress information, instead of stream order, to output results or purge state. In stream systems using the new OOP architecture, query operators can let tuples through on the fly, do not need to maintain stream order, and thus can avoid the associated costs. A short example comparing IOP and OOP follows.

1.1. An Example Comparing IOP vs. OOP

Consider the IOP evaluation on a query, Q1-2, from a network-monitoring scenario similar to those discussed in Gigascope applications [33].

Q1-2: “Count the number of packets from three links, Control, Main1 and Main2, for every minute; update the result every minute.”

SELECT count (*) [RANGE 1 minute, SLIDE 1 minute, WA ts]
FROM Control union Main1 union Main2

Q1-2 monitors streams of network packets arriving on three separate links and computes the number of packets received over a *tumbling window* of length 1 minute defined on window attribute (WA) *ts*. A tumbling window is a commonly used type of window for aggregation, with non-overlapping window extents. Packets from each link arrive in order of the timestamp attribute *ts*. The Control link contains almost no traffic; Main1 and Main2 are high-rate data links, and might not be synchronized with respect to their timestamp attributes because of variations in transmission delays. We note that streams with widely varying volumes and delays arise in various stream applications, including network-traffic monitoring, financial data processing, and intelligent transportation monitoring. In the rest of the thesis, we will use network-monitoring applications as our working scenario.

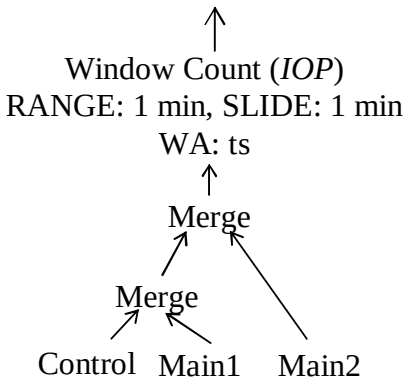


Figure 1-1 IOP qury evaluation for Q1-2

Consider the cost of enforcing order (for the intermediate streams) in the IOP architecture for Q1-2 with ordered inputs. Figure 1-1 shows an IOP query plan for Q1-2. Here, the implementation of the Window Count operator requires ordered streams in order to determine the boundaries of window extents. Consider the potential buffer-space requirements and tuple-processing delay resulting from enforcing order on the intermediate results feeding into the Window Count. The Merge operator is an order-preserving implementation of logical (bag) Union that combines the input streams and guarantees that the output stream is ordered. To do so, it may need to buffer a significant amount of data. For example, during lulls on the Control link, the Merge operators in Figure 1-1 have to buffer tuples from the Main links. Also, if there is timestamp (*ts*) skew between the links, the Merge operators will have to buffer tuples to synchronize the links. The exact amount of buffer space that the Merge operators require is a function of the arrival pattern of the input streams, such as duration of lulls, packet rates on the three links, and their timestamp skew, but there is no upper bound. In addition to the memory requirements, buffering also increases tuple latency. Notice that the overhead here for the IOP evaluation of Q1-2 is mostly for enforcing order on the combined stream, to satisfy the requirement of the Window Count operator.

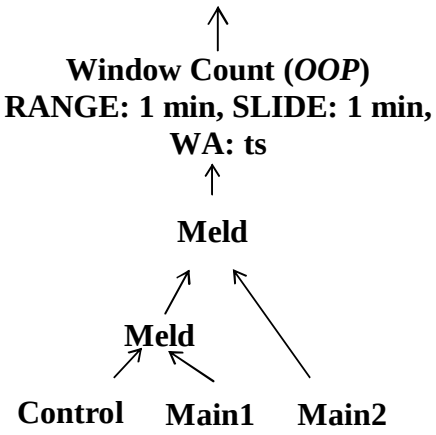


Figure 1-2 OOP query evaluation for Q1-2

In contrast, consider the OOP evaluation of Q1-2, which is shown in Figure 1-2. In its OOP evaluation, the aggregate operator uses an order-agnostic implementation, called WID [41], which we will present in detail in Chapter 6. WID views the window construct as a group-by condition on a function of the windowing attribute and relies on *punctuation* (i.e., a special type of tuple embedded in a stream to indicate the end of sub-streams) for end-of-window notification. (Note that as each input stream is ordered, it is easy to insert punctuation into the streams if it is not already there.) The OOP evaluation of Q1-2 replaces the order-preserving Merge with a simple Union that passes tuples through and buffers no tuples, which we call *Meld*. In addition, WID directly reduces tuples into partial aggregates. It immediately consumes input tuples, possibly maintaining partial aggregates for multiple "open" windows. These active aggregations are the only state that the OOP approach needs to maintain for Q1-2. Although the OOP approach may need to keep partial counts for multiple windows

when an input stream is late, in general, the required space is much less than would be required to buffer tuples.

1.2. The Scope of This Thesis

There are three main areas of contribution for this thesis.

First, we introduce a new data-stream model that does not assume ordered stream arrival and present a formal framework for explicitly defining window semantics, and then give definitions for existing types of window using this framework. In our definitions, window semantics is determined only by the window parameters and the windowing-attribute values of input tuples, regardless of physical data properties, such as data-arrival order. We also discuss the window semantics for join, independent of physical properties.

Second, based on this new window-semantics definition, we developed new, order-agnostic evaluation techniques for window aggregation, including WID (Window-ID), Paned-WID, and AdaptWID for windowed data-reducing operators. The first one is the basic order-agnostic implementation and the latter two are optimization of WID. In general, these new implementations can process input streams without relying on their arrival ordering, and need neither to buffer nor materialize window extents. We also propose order-agnostic implementations for two commonly used window joins, sliding-window join and tumbling-window join. Our experimental study shows that the new techniques significantly improve the overall performance of the evaluation of window operators compared to existing approaches.

Third, we propose a new architecture, OOP (Out-of-Order Processing), for stream-query evaluation. Our new techniques for windowed-operator implementation, which do not require ordered input streams, allow OOP evaluation of stream queries. Query operators in the OOP approach are freed from the burden of maintaining order, and thus the overall performance of query evaluation may be significantly improved. We discuss the OOP query evaluation approach and experimentally compare OOP versus IOP evaluation for stream queries, in particular, data-reducing stream queries. Our experimental results in two stream-processing systems show the benefits of the OOP strategy in memory usage and response latency, with at least comparable execution time.

Chapter 2

BACKGROUND

Stream-query evaluation is comparable to relational-database-query evaluation. A stream has a schema and is comparable to a relation in relational database. A relation is a set of tuples. (We use "relation" is in the broad sense that duplicates are allowed.) Relational query operators map relations to relations. For example, a Select operator takes a relation as input, applies a predicate to the relation, and produces a relation that contains only tuples in the original relation that satisfy the predicate. An Aggregate operator takes a relation and produces an aggregate value, which can be viewed as a special relation with a single tuple. An Aggrgeate operator with group-by attributes takes a relation and produces a relation with a tuple for each group defined by the grouping attributes—each tuple in the result relation contains an aggregate value and the grouping attribute values. The Join operator takes two relations, applies a join predicate, and produces a relation that contains pairs of tuples that match via the join prediate. In relational database systems, a query operator can have multiple physical implementations, and thus a "logical" query operator may correspond to multiple "phyiscal" query operators. (For example, the Join operator can have multiple implementations such as Hash Join and Nested-Loops Join.) To evaluate a query, relational database systems need to translate a logical query into a query execution plan that consists of physical query operators. As each operator may have multiple implementations, relational database systems use a *query optimizer* to pick an

optimized query execution plan from a set of possible execution plans, based on the logical query itself, the physical properties of the relations involved in the query, and an expected-cost model for plan evaluation.

Stream-query evaluation conceptually resembles relational-query evaluation in many ways, but it presents a different set of challenges. Unlike a relation that contains relatively static data, in a stream, tuples arrives continuously and stream systems have no control over the arrival rate, order and pattern. Examples include bid streams in online-auction monitoring systems and network-packet streams in network monitoring systems. A stream query is comparable to a relational-database query – it consists of query operators similar to relational query operators, as we have seen in Chapter 1. However, as streams are potentially unbounded, in stream queries, blocking query operators (e.g., aggregation) and stateful query operators (e.g., join) often need a window condiditon so that they can output results and purge state. Query operators implemented in a pipelined way, such as Select and Project, can be adapted to stream queries easily.

We believe that important issues for (windowed) stream-query evaluation include a lack of logical semantic definitions for window stream-query operators that are independent of physical stream-arrival properties such as arrival rate and order, handling stream "abnormalities", and the performance of stream-query evaluation.

First, as we will show, a logical semantic definition of window-query operators will form the basis for a flexibile and scalable stream-system implementation, as well as for the optimization of stream-query evaluation. Just as in relational databases, the

logical definition of relational-query semantics that is independent of physical storage properties of the stored relations, allows different implementations of the same logical query. In the literature, the semantics of window operators are usually described operationally and assume ordered and continuous streams. However, in real applications, stream "abnormalities" such as out-of-order tuples and lulls arise naturally. We present our definition for window semantics that are independent of physicial stream-arrival properties in Chapter 5.

Second, stream systems have no control over their physical-arrival properties such as arrival order, arrival rate and arrival pattern, and thus they have to deal with various stream "abnormalities". These situations can lead to great burdens in the implementation of stream systems and overhead in stream-query evalution, but they arise naturally in stream applications. For example, join may produce disordered results even when its input streams are ordered and synchronized; combining two ordered streams can lead to disorder, unless the streams are exactly synchronized. Many stream systems need to ensure the order of inter-operator intermediate streams, which increases the complexity of the implementation of the stream system and limits possible optimizations. Such systems also need to sort disordered input streams, which increases the performance overhead of stream-query evaluation. Highly selective predicates in a Select or Join operator can filter out most tuples from a stream and lead to stalls in the operators following the operator with the selective predicate. Delays in transmission may produce lulls on an input stream, which may also stall query execution. In this thesis, we present implementations of window operators that deal

with stream abnormalities naturally and also perform better than existing implementations. We also present a new architecture for stream-query evauation systems that "glues" our new operator implementations together, and provides better performance in stream query evaluation at the system level.

Third, the performance of stream query evaluation must satisfy the needs of stream applications, and is normally evaluated from three perspectives: memory requirements, CPU cost, and latency. As streams may contain large amounts of data and are potentially unbounded, it is always important to keep the memory usage of stream-query evaluation low and ensure that it does not grow without bound with data arrival from input streams. The CPU cost of stream-query evaluation determines the capacity of a stream system—the maximum stream rate that a system can sustain. Also, many stream-monitoring applications have (near) real-time requirements, and thus the latency of query results is an important performance measure for stream-query evaluation.

In the following, we first review the punctuation mechanism. All the work presented in this thesis leverages this mechanism. Then, we review the architecture of two stream systems—NiagaraST, a stream query engine that we built by extending the Niagara Internet query system developed at the University of Wisconsin–Madison [46], and Gigascope, a network-packet monitoring system developed at AT&T to monitor their backbone network [14].

2.1. Punctuation

Punctuation is a general mechanism proposed for indicating the ends of sub-streams. A punctuation is a special tuple embedded in a stream, having the same schema as normal tuples in the stream. Bounded sub-streams can allow blocking operators to produce results at the ends of sub-streams and stateful operators to purge the state for each sub-stream when it ends. For example, consider a network-packet stream, S, with schema <*srcIP*, *destIP*, *srcPort*, *destPort*, *len*, *ts*>. The punctuation p_1, (202.3.12.4, *, *, *, *, *), embedded in S indicates that there are no more packets with source IP 202.3.12.4 in the network-packet stream following p_1. For an aggregate query evaluated over S that computes the count of packets from each source IP, the query can output the count for source IP 202.3.12.4 upon receiving p_1. Punctuation can have multiple punctuating attributes and provide predicates with various patterns. For example, the punctuation p_2, (202.3.12.4, *, *, *, *, (, 12:00:00AM)), has two punctuating attributes, *srcIP* and *ts*, and indicates that there are no more packets with source IP 202.3.12.4 and timestamp smaller than 12:00:00AM. Here, in punctuation p_2, the pattern (, 12:00:00AM) is a range predicate that matches all the packets with timestamp values from the the semi-bounded interval up to 12:00:00AM. Previous studies on stream-query evaluation consider leveraging punctuation to allow early output of results from blocking operators and to reduce state maintained by stateful operators [15, 39]. Gigascope uses punctuation to handle lulls, for example, to unblock the order-enforcing implementation of Union during lulls on a low-volume network-traffic link.

In this work, we use punctuation to communicate the progress of data streams (both input streams and inter-operator streams) to query operators. For example, punctuation p_3, (*, *, *, *, *, (, 12:00:00AM)), in a network-packet stream shown in Figure 3-1 indicates the current stream low-watermark is 12:00:00AM, which means all packets with *ts* attribute value smaller than 12:00:00AM have arrived.

(202.1.3.0, 202.2.1.2, 5, 10, 102, 11:59:23PM)
(102.3.2.7, 211.9.3.6, 10, 10, 200, 11:58:00PM)
(202.1.3.0, 202.2.1.2, 5, 10, 111, 11:59:53PM)
(202.6.9.2, 211.7.3.1, 11, 9, 300, 11:59:55PM)
(*, *, *, *, *, (, 12:00:00AM)) $\boldsymbol{p_3}$
(211.1.3.0, 202.2.1.2, 5, 10, 100, 12:00:23AM)
(202.3.1.1, 102.5.0.2, 9, 11, 210, 12:00:53AM)
⋮

Figure 2-1 Punctuation p3 embedded in a network-packet stream.

In this thesis, we generally assume *linear punctuation*—a special case of punctuation that uses an ordering attribute (e.g., the windowing attribute for window operators) as a punctuating attribute and for which the value of the ordering attribute in the punctuations in the stream is monotonic. If punctuation also contains other data attributes, such as the grouping attributes of an aggregate operator or joining attributes of a join operator, the ordering attribute must be monotonic within each group—we term this type of punctuation *group-wise linear punctuation*. For example, a stream may contain punctuation on the *ts* attribute for data from each source IP, and if punctuation for each source IP has monotonicly increasing *ts* values, it is group-wise linear punctuation. Compared to linear punctuation, group-wise linear punctuation

provides stream progress at a finer granularity, and thus potentially allows earlier output and more efficient state management. Rules for query operators for processing and propagating punctuation have been studied previously [71]. We will also discuss the implementation of the punctuation rules when we present our implementations of stream query operators. Hereafter, since we only consider linear or group-wise linear punctuation, we use a single value, instead of a range value, for the ordering attribute value in punctuation. For example, we will use the punctuation (102.2.45.10, *, *, *, *, 12:00:00AM) instead of (102.2.45.10, *, *, *, *, (, 12:00:00AM)) to indicate that all packets from IP 102.2.45.10 with *ts* attribute value no greater than 12:00:00AM have arrived.

2.2. NiagaraST

NiagaraST is a stream query engine that we built by extending the Niagara Internet Query System developed at the University of Wisconsin–Madison [46]. Niagara is a pipelined, push-based query system written in Java that supports XML-format data. In Niagara, query operators are implemented as OS-scheduled threads, and query operators in a query plan are connected with data queues. Every query operator waits on its input queue(s), and puts results to its output queue(s). The queues actually contain pages of tuples rather individual tuples—an operator writes to its output queue once it has produced a page of tuples. (The default size for a page is 30 tuples.)

Having data pages in queues, instead of individual tuples, reduces the cost of

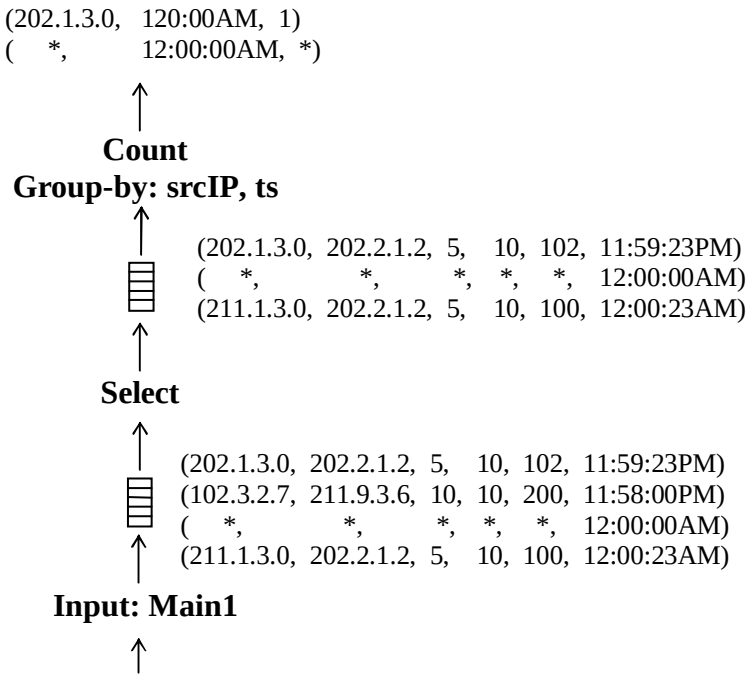


Figure 2-2 Query plan for Q2-1 in NiagaraST

synchronization of operator threads. NiagaraST inherits the system architecture and query execution model of Niagara. Figure 3-2 shows the query plan for the following query, Q2-1, in NiagaraST. Q2-1 computes the number of packets from each source IP for every minute, as the *ts* attribute is given in minutes. In Figure 3-2, the Select operator filters out packets whose srcIP does not match the given two source IPs. The Count operator maintains a hash table to compute the number of the packets from the two source IPs in each minute. Punctuation indicates the end of each minute and allows the Count operator to output results and purge its hash table.

Q2-1: "Count the number of network packets in the Main1 link from source IP 202.101.0.0 and 202.101.0.1 in every minute."

```
SELECT      srcIP, ts, count (*)
FROM        Main1
WHERE       srcIP = "202.101.0.0" or srcIP = "202.101.0.1"
GROUP BY    srcIP, ts
```

In NiagaraST, we enhanced the original query operators in Niagara with punctuations and also added new query operators, such as window aggregation and windowed join, to support queries over streams. Leveraging punctuation to express the progress of streams, NiagaraST does not rely on ordered streams in its evaluation of windowed queries.

2.3. Gigascope

Gigascope is a stream system developed at AT&T that monitors network traffic in AT&T backbone networks [14]. It is written in C and C++. Gigascope supports a

SQL-like language, GSQL. It is a generated-code system—users write queries in GSQL and the queries are translated into C and C++ code, which is then compiled into executables. Gigascope supports tumbling-window aggregation natively, and can support sliding-window aggregation via user-defined functions. Gigascope also supports join with an equality predicate on a monotonically increasing timestamp attribute.

Network-traffic-monitoring applications often need to reduce network-traffic streams into aggregated forms such as NetFlows (i.e., records summarizing network connections) before further processing. Thus, aggregation is critical for the efficiency of network-monitoring stream queries. In Gigascope, an aggregation query is split into a light-weight, low-level aggregation that significantly reduces the data volume, and a high-level aggregation that rolls up the results of low-level aggregation. The grouping and windowing conditions of the corresponding low-level and high-level aggregation are the same. The low-level aggregation directly processes input packets from a ring buffer and maintains a hash table to incrementally compute aggregates. The size of the hash table is fixed, so that the low-level aggregation can process packets fast enough to keep up with the speed of network traffic. However, the low-level aggregation has to output an existing partial aggregate on hash collision. Low-level aggregation and high-level aggregation execute in different processes, as low-level aggregation can potentially run on a different machine or be pushed down to the network-interface card on a router. In Gigascope deployments, the data volume that the join operator needs to handle under normal conditions is much lower than what aggregate operators need to

handle, and the joining condition is very selective; thus the data volume of the results produced by join is relatively low.

Gigascope preserves stream order in query evaluation. In order to handle lulls, it also supports linear punctuation [33]. Gigascope may need to evaluate queries over the combined stream of very high-volume main-network traffic links and relatively low-volume control links, which naturally introduce lulls. Lulls on the control links block the evaluation of stream queries and increase their memory usage. We have discussed this situation in the example in Section 1.2. To deal with this issue, during the lulls on the low-volume control lulls, Gigascope estimates the progress of those control links on the timestamp attribute of the packet tuples, and inserts linear punctuation on the timestamp attribute that carries the progress information into the packet streams. During lulls, punctuation helps to unblock query operators that block on the low-volume control links and may also help these operators to purge state on the fast main links.

Chapter 3

RELATED WORK

Windowing is not unique to stream queries. Windowing is used with aggregtion in relational database systems. SQL-99 defines a window clause for use on stored data, and many database vendors support the window clause. SQL-99 limits windows to sliding by each tuple (i.e., each tuple defines a window extent), thus tying each output tuple to an input tuple. In comparison, in stream queries, the spacing of the consecutive window extents is normally specified by users in terms of domain values such as time interval or tuple sequence numbers [61]. This type of window is more suitable for applications with bursty or high-volume data. For example, in network-monitoring applications, one possibly wants network statistics updated at regular intervals, independent of surges or dips in traffic. Also, getting a result tuple per input packet may overwhelm network-monitoring applications. SQL-99 allows a window to extend to one or both sides of the target input tuple, while in stream queries, a window normally only extends backward (descending timestamp or sequence number)—extending forward would require knowledge of future data.

In this chapter, we review the evaluation of window aggregation and window join in the literature, as well as disorder-handling mechanisms presented before. We also briefly discuss other stream query systems.

3.1. Window Aggregation Implementations

In the following, we review research work in three categories in the literature that relates to the three types of window aggregation evaluation that we will present in Chapter 6, including the basic evaluation technique for window aggregation, the evaluation of window aggregation using shared sub-aggregation, and adaptivity in query evaluation.

A common method for evaluating window aggregation is the *buffering* tech1nique. It requires ordered input streams (or enforcing order on the input streams), materializes each window extent, and continuously applies the appropriate relational query operator over each materialized extent. Previous implementations of window aggregation more or less resemble the buffered technique. Arasu et al. [3] model and implement window operators as relational query operators over instantaneous relations whose content changes with the arrival of new tuples and expiration of old tuples. Aurora [4] enforces the ordering of input streams and can support windows by user-defined functions. Gigascope [14] supports tumbling-window aggregates by grouping on a function over the timestamp attribute, similar to the *wids*() function in our WID implementation that will be presented in Section 5, but which requires ordered input streams to unblock the aggregate operator. The *negative-tuple* approach, which sends tuples with a special "negative" flag to query operators to indicate that they are expired, has been introduced for windowed aggregate-operator evaluation [3, 10, 26]. With negative tuples, the aggregate for the next window extent can be initialized based on the aggregate for the current window extent then adjusted using the negative tuples.

Thus, the negative-tuple approach can reduce the computation of aggregation for the overlapping parts of consecutive window extents. The negative-tuple approach is complementary to the WID implementation. For example, the super-aggregation of the paned-WID implementation as discussed in Section 6.2 can leverage negative tuples to further reduce computation.

Computing and sharing sub-aggregation is a common technique for improving the computational efficiency of query evaluation in general. In relational database systems, the sub-aggregation and super-aggregation concept is used by the ROLLUP operator in SQL-99 and the data cube operator [23] to express a set of aggregates at different granularities. These operators provide an efficient and readable way to express aggregation along a hierarchy—for example, city, state, and country—but are used over stored data.

Sub-aggregation sharing is also adopted in the evaluation of stream aggregate queries. The paned-WID implementation as discussed in Section 6.2 shares sub-aggregation among consecutive window extents. In Gigascope, the evaluation of tumbling-window holistic aggregates (e.g., quantile and heavy-hitter) uses fast, lightweight sub-aggregation for early data reduction followed by super-aggregation in which expensive processing is performed [13]. However, Gigascope does not share sub-aggregation among multiple window extents, as it assumes non-overlapping extents (tumbling windows). Zhang et al. [82] share fine-granularity sub-aggregation for multiple coarse-granularity stream aggregate queries (i.e., they use aggregation with finer groups to compute aggregation with coarser groups). Arasu and Widom [5]

propose two algorithms, B-Int and L-Int, for shared execution of multiple sliding-window aggregates with different window sizes. Their algorithms support a user-polling output model. They maintain a data-structure that stores the sub-aggregates over the active part of the stream at many different granularities. When a user polls a query, the aggregate for the latest complete window is computed by looking up the constituent sub-aggregates stored in the data-structure, and aggregating those values. Both B-Int and L-Int reduce computation cost, but at the cost of increased memory space. Krishnamurthy et al. propose to use aggregation of "paired windows", which are similar to panes, for shared execution of multiple window-aggregation queries [37]. Nagaraj et al. [47] propose a sub-aggregate-sharing technique that shares intermediate sub-aggregates among multiple stream-aggregate queries; their work uses a computation cost model to select the set of minimum-cost intermediate sub-aggregates that cover the target aggregates [47].

AdaptWID is an adaptive implementation for window aggregation that deals with the excessive memory usage induced by data-distribution skew. Adaptivity is a very broad term in the context of query processing. Both relational database systems and stream query systems leverage adaptivity to optimize resource usage. One class of adaptivity used for processing both static and streaming data is query-plan re-optimization, where operators in a query plan are reordered or changed based on updated information or changing conditions [4, 34, 75]. Another class of adaptivity for processing streaming data is exemplified by the Eddies project [6], in which the route a tuple takes through operators is determined dynamically based on operator

selectivities, input rates and operator costs to improve system throughput. A final class of adaptive approaches used for both static and streaming data is operators that adapt their state to data characteristics. For example, XJoin [73] may store data to disk if the stream arrival rate exceeds its processing capacity and then adapts between processing current streaming data and processing previously stored data based on the arrival rate of the input streams. MJoin [76] processes multiple input streams and adapts the join order based on the availability of the inputs. The rate-based optimization of Kang, et al. allocates memory to operators in proportion to stream speed, assuming stream speed is known at optimization time. Aggregation in Gigascope may adapt from keeping exact aggregates to maintaining approximate sketches [13]. AdaptWID differs from these adaptive techniques in that our algorithm adapts the behavior (state and query processing) of the aggregate operator to data skew, but still gurantees exect answer.

3.2. Window-Join Implementations

Sliding-window join and tumbling-window join are the most extensively discussed stream-join operators in the literature. The window condition with a join is a join predicate defined on the windowing attribute. Query Q3-1 is a sliding-window join example, defined on the windowing attribute, *ts*, with a 3-minute window on the first input and a 2-minute window on the second input. This join specifies that a tuple, *l*, from the first input, joins with tuples with *ts* value greater than (*l.ts* – 2 min) from the

second input, and that a tuple, *r*, from the second input joins with tuples with *ts* value greater than (*r.ts* – 3 min) from the first input.

Q3-1: "Find network packets pairs from Main1 and Main2 links; the source IP of the Main1 packet should match the destination IP of the Main2 packet; the *ts* attribute of the Main1 packet should be no more than 2 minutes later than the *ts* attribute of the Main2 packet and the *ts* attribute of the Main2 packet should be no more than 3 minutes later than the *ts* attribute of the Main1 packet."

```
SELECT *
FROM Main1 [WA ts, RANGE 3 min],
        Main2 [WA ts, RANGE 2 min]
WHERE Main1.srcIP = Main2.destIP;
```

In general, pipelined join implementations used in relational databases, such as symmetric hash join and symmetric nested-loops join, can be adapted for use with streams by adding a state-purging strategy. Most previous work on sliding-window join assumes that windows are defined on arrival time [15, 21, 26, 35, 62], or that input streams of the join arrive ordered and synchronized on a shared timestamp attribute. This assumption implies that tuples from both streams share a "global order" – the timestamp of a new tuple is guaranteed to be no smaller than that of any tuple already arrived from either input stream. Based on this assumption, a window-join implementation maintains a "window" of tuples for each input stream. For example, for query Q3-1, the last 3-minutes of tuples are maintained for the Main1 input and the last 2-minutes of tuples are maintained for the Main2 input. When a new tuple arrives, join can purge state based on the timestamp of the new tuple. For example, when the join operator of Q3-1 receives a new tuple, *l*, from Main1, it purges Main2 tuples with *ts* value smaller than *l.ts* – 2 min. Then, *l* is matched with stored tuples of Main2, and

composite tuples of *l* and the matching tuples are produced, and then *l* is stored. However, if the global-order assumption is not satisfied, this window-join evaluation might produce incorrect results. For example, suppose Main2 tuples arrive one minute later than Main1 tuples. That is, a tuple, *l*, in Main1 arrives approximately together with a tuple *s* in Main2 with *s.ts* = (*l.ts* – 1 min). When *s* arrives, tuples in Main1 with *ts* value smaller than (*s.ts* – 2 min) have been purged by *l*, and thus *s* will not be matched with the Main1 tuples with *ts* values between (*s.ts* – 3 min) and (*s.ts* – 2 min). So, part of the results will be missing in this case. Also, when *l* arrives, tuples in Main2 with *ts* value smaller than (*l.ts* – 2 min) are still maintained and will be matched with *l*, and thus incorrect results may be produced.

Hammad et al. [27] propose sliding-window implementations that support ordered input streams, but with potential arrival-time skew, and analyze the average response time of those implementations. To optimize the output rate of a window join, Kang et al. [35] propose asymmetric join implementations that can process each input stream of the join individually with nested-loops join or hash join, based on the relative arrival rates of inputs. Hammad et al. [27] propose scheduling schemes that optimize for different metrics, such as maximum throughput or shortest-window first, for shared execution of multiple sliding-window join queries. Ding and Rundensteiner [15] exploit punctuation on data attributes—instead of on the windowing attribute—for aggressively purging state by data attributes, to reduce memory usage of window join queries. Srivastava and Widom [62] present algorithms for producing approximate answers for sliding-window join with limited memory resources. Golab and Ozsu [21]

propose lazy probing and purging to improve computational efficiency for the evaluation of sliding-window multi-joins that execute multiple joins together as a series of nested-loop joins.

3.3. Handling Disorder in Streams

Out-of-order data is one of the important challenges that stream-processing systems need to handle [66]. A common way handling out-of-order data is to sort the data. Slack [4] and heartbeats [61] are two mechanisms proposed dealing with disorder in stream systems. *Slack* is a parameter specified by the user as a number of tuples or a timestamp value that indicates the maximum amount of disorder allowed in a stream. A query operator takes a slack parameter and deals with disorder by buffering as many most-recently arrived tuples as specified by the slack parameter. These buffered tuples are sorted and thus, as long as the input disorder is within the slack amount, the query operator can still process tuples in order. Aurora introduces the *BSort* operator, which is a slack-aware sort operator. Other query operators in Aurora may also take a slack parameter and may handle disorder themselves.

Heartbeats are also used for dealing with disorder. A *heartbeat* is a type of control signal used to indicate the arrival of input tuples in terms of their timestamp attribute values. Heartbeats also represent the advancing of time in the absence of tuples (i.e., during lulls). Heartbeats are similar to punctuation on the timestamp attribute, but punctuation is more expressive. In addition, rules have been formally defined for how

query operators should propagate punctuation, while heartbeats are an ad hoc mechanism.

A few stream systems allow out-of-order tuples. Borealis proposes revision mechanisms that process delayed tuples as insertions; these revision mechanisms support the processing of streams with limited disorder [2]. The Juggle operator [50] from the Telegraph system intentionally reorders a data stream in order to advance "interesting" tuples. Mazzucco et al. [45] consider a key-based merging algorithm (actually, a join) for high-volume data streams that copes with large amounts of disorder by dropping tuples or using approximate matches. Hwang et al. [30] propose punctuation-aware, order-insensitive implementations for window aggregation and window join. However, these order-insensitive operators are relatively heavyweight, and are designed for latency reduction in a low-throughput system. In contrast to other approaches for disorder handling that deal with disorder at the operator level, our OOP approach deals with disorder at system level—it leverages punctuation to communicate the progress of streams and thus allows query operators to be order agnostic. By avoiding maintenance of stream order, OOP exhibits large advantages in terms of limiting memory usage, reducing latency, smoothing workload and thus OOP significantly improves query evaluation throughtput.

3.4. Other Stream-Query Systems

In a broad sense, my thesis work relates to the field of stream-query evaluation in general. Several on-going and completed research projects have been working on

stream-query evaluation. The STREAM project focuses on the general semantics of stream queries and the theoretical analysis of the memory efficiency of stream-query evaluation [63]. TelegraphCQ is a stream query engine that adapts tuple processing based on the system workload [12]. Law et al. propose to use user-defined aggregates (UDA) in SQL to deal with the issue of blocking operators [38]. With UDAs, users can explicitly specifiy when and under what conditions an Aggregate operator should produce results. The Aurora stream-query engine can provide quality-of-service support for stream query evaluation and sheds load with respect to users' QoS requirements [4]. Borealis is a distributed stream-processing engine [2], built on Aurora and Medusa, a distributed system infrastructure [81]. Borealis focuses on distributed, scalable, and fault-tolerant stream processing [29, 30, 68, 80]. Gigascope supports network-monitoring applications and focuses on processing high-volume network-packet streams at line speed [27]. CEDR focuses on providing flexible latency-accuracy tradeoffs for stream-query evaluation—higher accuracy may incur more latency [8]. CEDR query operators may produce "preliminary" results to reduce latency; these query operators may also retract previously released preliminary results later and may produce "revisions" to replace the retracted results. Query operators in CEDR are able to naturally process regular tuples, as well as retractions. Other prototype-stream query engines include Nile, which integrates certain online data-mining functionality with stream-query processing [18, 28], AQSIOS, which focuses on the secheduling of multiple stream queries [51, 52, 53], and CAPE, which leverages punctuations in its stream-query evaluation [15, 16, 77]. System S is a

distributed stream processing engine developed at IBM [20]. It supports a programming language called SPADE, which supports generic data types and building-block operations as well as stream-query operators, and a framework for developing stream applications. System S supports various types of streams such as financial records and sensor data through “input adapters”.

Previous research on sequence databases, temporal databases, active databases, and incremental maintenance of materialized views is related to stream queries, as that research either involves query operators over special attributes from ordered domains, or requires continuous query evaluation. Sequence databases support efficient expression and evaluation of queries over data with attributes from ordered domains, such as timestamps and positions [56, 57, 58]. Temporal databases maintain all database states, instead of the current snapshot, and they support queries on data’s valid times—the time that the data are “alive” [55]. Some active databases [54]monitor append-only tables and trigger active rule re-evaluation upon tuple insertion [54]. Incremental maintenance of materialized views typically requires incremental query evaluation with one-pass algorithms that need at most just one scan of the data, so that view maintainance can be efficient. The Chronicle data model is proposed to define a constraint language that can only allow views that are incrementally maintainable [32]. All these research efforts deal with the situation where data is well organized and controlled. Although the type of queries that stream systems support may seem similar to what are supported by such previous research, stream-query evaluation presents a different set of challenges, as streams arrive

continuously and are potentially unbounded, and stream applications typically have (near) real-time requirements. For example, stream systems require low memory and low latency algorithms and may have to handle stream abnormalities such as disorder and lulls.

Chapter 4

PROGRESSING STREAMS

Part of the motivation of this thesis is to separate the notion of progress of a data stream from its physical arrival order and thus allow more flexibility in the implementation of query operators. In this chapter, we present a new data model for streams, the progressing-stream model, which introduces the notion of stream progress and relaxes the ordering requirement that many stream systems assume. Intuitively, stream progress is defined on an ordering attribute and describes the arrival of a stream in terms of the ordering-attribute value. Punctuation can express stream progress naturally. For example, the punctuation *p*3, (*, *, *, *, *, 12:00:00AM), in Section 2.1 indicates that the network-packet stream has progressed to 12:00:00AM according to its *ts* attribute. In this chapter, we present only the conceptual stream model, and leave the discussion of the implications of this model on stream-system implementation to later chapters.

The progressing-stream model is in direct contrast with the commonly used model of a data stream as an ordered sequence of tuples. In IOP systems, stream-query operators rely on stream order to determine when to output results for blocking operators and when to purge state for stateful operators. The key observation motivating the progressing-stream model is that although IOP stream systems rely on ordered streams to unblock and purge, total order on an attribute is not required. Instead, any operator that can be unblocked and purged using an ordered attribute can also be handled with a

progressing attribute, as long as we can detect and communicate stream progress. The benefit of the progressing-stream model is that it separates stream progress from physical arrival of stream tuples, and thereby allows more flexibility in implementations of stream query operators. In the following, we present the stream-progress model.

In the *progessing-stream model*, we model a stream as a sequence of tuples that "progresses" on a data attribute, *A*. That is, the value of the *A* attribute in the stream always eventually exceeds any fixed value *v*. We term attribute *A* the *progressing attribute* of the stream, and assume that *A*'s domain is discrete. In practice, the progressing attribute is often a timestamp of some form. Potentially, *A* can be any tuple attribute with an ordered domain, and thus stream systems can use either timestamps assigned by external data sources or internally by the system as the progressing attribute.

To define the notion of progressing stream, we first define the low-watermark (*lwm*) for attribute *A* of stream *S* at *n*. Let S_n be the prefix of S of length *n*. Then,

$$lwm(n, S, A) = \min\{t.A \mid t \in S - S_n\}. \qquad \text{(Eq. 4.1)}$$

That is, *lwm*(*n*, *S*, *A*) is the smallest value for *A* that occurs after the prefix S_n of stream *S*. Intuitively, the low-watermark indicates the progress of stream *S*—the low-watermark at *n* indicates the smallest value that may occur after S_n in *S*. (Thus, the largest value for *A* that will not occur after S_n in *S* can also be derived.) For example, suppose *S* contains tuples t_1, t_2, t_3, t_4 and so on, and each tuple contains a timestamp attribute value; if the low-watermark at t_4 is 10:00:00AM, it means that there are no

tuples arriving after t_4 that have a timestamp smaller than 10:00:00AM. Figure 4-1 shows the low-watermark of a disordered stresam. In general, low-watermark cannot be computed based on past tuples—low-watermark potentially requires global information. However, in practice, we can insert punctuation into *S* to explicitly communicate bounds on the low-watermark.

Definition: Stream *S* is *progressing on attribute A* if for every value *v* in the domain of *A*, there exists an *n* such that $lwm(n, S, A) > v$. When this condition holds, we say *A* is a *progressing attribute* for *S*, and that *S* is a *progressing stream*.

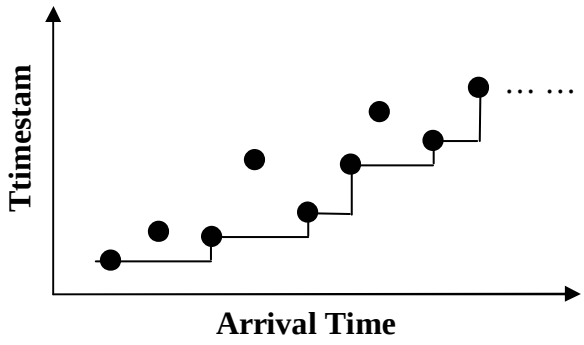


Figure 4-1 Low-watermark (lwm) of a disordered stream that progresses on the timestamp attribute

Previous work on data streams commonly models a stream as a potentially unbounded sequence of data items arriving in order. However, modeling a data stream as an ordered sequence of tuples conflicts with the reality that stream disorder occurs naturally in real-world stream systems. The following are a few causes of stream disorder.

- Items arriving over a network from a remote origin may take different paths with different delays.
- In a parallel or distributed system, a data stream may be a combination of several sub-streams from different nodes. The merged stream can be disordered if there are different processing or transmission delays associated with those nodes.
- Some data streams have multiple timestamp attributes with different orders. For example, NetFlow [48] records from a router might arrive in order of "flow end" time, but are disordered on "flow start" time. Some queries may window on "flow end" and others on "flow start."
- Even when data streams arrive in order, some query operators, such as sliding-window join, can introduce disorder in intermediate results.
- Data prioritization [74] may also cause disorder.

Note that although streams are disordered, a progressing attribute exists for each of the cases above. For the first and the second example, the data items' timestamp from their data source is the progressing attribute; for the third example, either "flow start" time or "flow end" time can be the progressing attribute; for the fourth example, the timestamp from either input stream can be the progressing attribute; for the fifth example, the progressing attribute stays the same after data prioritization. We believe that our progressing-stream model better represents real-world data streams. The benefit of having a progressing attribute and knowing the progress of a stream is that

window operators defined on the progressing attribute can incrementally produce result and be incrementally purged without requiring ordered streams.

With the progressing-stream model, we have a remaining issue: How do operators get progress information on streams? Even if a stream is progressing, that does not actually tell us the progress at every point in time. We will address this issue in Chapter 6, which presents the implementation of stream-query operators. Before that, we present the order-agonistic semantics of some stream-query operators in the next chapter.

Chapter 5

WINDOW SEMANTICS

As we discussed in Chapter 1, lack of an explicit definition of window semantics for window operators leads to confusion and inefficiency in the implementations of those operators. In this chapter, we present a formal definition of window semantics for window aggregates, and also discuss the semantics of window join. Note that for windowed query operators, we assume that the window is always specified on one of the stream's progressing attributes, and thus a progressing attribute is also called a windowing attribute here.

Window operators support new user requirements and address the limitations of traditional query operators when used over streams. Users of stream-query systems are often more interested in querying recent data in the stream and having the query results updated periodically than getting information over the entire past stream. Traditional query operators only support one-time evaluation and do not provide such functionality. Further, traditional query operators are defined over a static relation and may not be applicable to potentially unbounded streams. For example, a blocking operator (e.g., aggregation) on relations normally requires the entire input data set before producing any results; and a stateful operator (e.g., join) may need to maintain an unbounded amount of state when the input stream is potentially unbounded. A window operator breaks the input stream into bounded sub-streams and evaluates the corresponding query operator over each sub-stream, and thus unblocks the operator

and limits the amount of state that the operator needs to maintain. The window condition of an aggregate operator is defined with a *window specification* consisting of a set of parameters, such as RANGE, SLIDE and WA. The window specification defines potentially overlapping finite sub-streams over an input stream. We call each finite sub-stream a *window extent*, and one or more aggregates are computed for each extent. For example, for the following query Q5-1, which is the same as query Q1-1 in Chapter 1, the window extents are 10-minute sub-streams that overlap by nine minutes. A network-traffic monitoring system can use such a windowed aggregate query to count the number of packets from each source IP in a link, M, for the past 10 minutes, advancing at 1-minute intervals. As before, the schema of the packets in M is *<srcIP, srcPort, destIP, destPort, len, ts>*.

Q5-1: "Count the number of packets from each source IP in the Main link for the past 10 minutes; update the results every minute."

```
Select srcIP, count(*) [RANGE 10 minutes, SLIDE 1 minute, WATTR ts]
From Main
Group By srcIP
```

There are different types of windows for aggregation. Q5-1 uses a *time-based sliding-window*, which is common in stream queries. We refer to the window as time-based because the windowing attribute is a timestamp attribute. We will also discuss other common window types later in this chapter, such as *tuple-based* windows that are defined on the tuples' arrival order, and *partitioned* windows that use a partitioning attribute to "split" a stream into partial streams before dividing each into window extents.

A fundamental problem with previous stream-query evaluation approaches is the lack of a logical definition for window operators. Logical definitions of query operators, independent of the physical properties of data and data storage and the particular algorithm used, are one of the most important advantages of relational database systems. Logical definitions of query operators allow users to focus on the meaning of their queries, regardless of physical data properties, and provide guidelines for the correctness of alternative implementations that optimize for different physical properties. Such logical independence of query operators is also important for stream-query operators. As we will discuss later, the logical definition of the window operators opens the way to more flexible and efficient implementations.

Previous approaches for implementing window operators generally require processing input tuples in windowing-attribute order, partly because they rely on ordered input streams to "operationally" determine window semantics. In general, we find these previous approaches to be inflexible and inefficient. Recall the buffering technique described in Section 3.1 that has been commonly-used previously. A typical buffered technique maintains input tuples in a window buffer, and determines window-extent boundaries based on the assumption that tuples are ordered. When the end of an extent is detected, the buffering technique computes the aggregate over the buffered tuples, which are exactly the content of the window extent, and then purges expired tuples from the buffer. Notice that the input stream must be ordered so that the content of window extents can be correctly determined with this technique. In addition, the buffered technique potentially requires large amounts of memory, as it buffers window

extents and may need additional space to enforce the order of the input stream. Buffered techniques also require progress of the input stream to guarantee progress of query execution. When there are too few tuples in the input stream, additional mechanisms such as timeouts and punctuations are needed to ensure progress of query execution. In summary, buffering techniques rely on the physical stream-arrival properties, that is, strict order, and continuous arrival. Stream imperfections, such as out-of-order tuples and lulls, must be handled as exceptions using additional mechanisms.

In this chapter, we present explicit logical definitions for window aggregation and join. In the next chapter, we will present implementations for window operators based on our definitions.

5.1. Window Aggregation: WID Window Semantics

In general, the window semantics of window aggregation is the relationship between tuples and window extents. In our definition, window semantics is determined by the window specification of the window aggregate and the windowing-attribute value of the tuples in the stream, and is independent of physical stream properties such as stream order and continuousness. It is also independent of any specific implementation of window aggregation.

Using Q5-1 as example, we show that window semantics can be defined independent of stream-arrival order. Consider the window extent *w* for Q5-1 corresponding to 10:10:00 AM – 10:20:00 AM. The content of *w* includes all the input tuples with *ts*

value within the range [10:10:00 AM, 10:20:00 AM). In general, the content of a window extent is independent of the arrival order of the input tuples, unless the windowing attribute is arrival time.

In the following, we first present a framework—the WID semantics framework—that consists of three functions for defining window semantics for window aggregation. Then, we present the window semantics definitions for various types of windows. Our window semantics is defined solely using the window specification and the values of the windowing attribute of the tuples in the input stream.

5.1.1. WID Semantics Framework

The WID semantics framework consists of three functions: *windows*(), which specifies the window-ids (i.e., window identifiers) for identifying window extents, and *extent*() and *wids*(), which define the mappings between window-ids and input tuples in either direction. All the three functions are derived from an operator window specification, *S*, and the set of tuples, *T*, in the input stream. Notice that *T* is only a logical entity and is not required to be materialized in our implementation for any type of window.

The *windows*() function defines a set of window-ids *W*, given a window specification *S* and a set of tuples *T*: *windows*(*S*, *T*) = *W*. We can use values from an ordered domain, such as non-negative integers, as window-ids. Suppose that the start point of Q5-1 is 00:00 AM and that we use non-negative integers for window-ids, then the first fifteen window extents can be identified with window-ids 0 – 14 as follows:

Window Extent	Window-Id
00:00:00 AM – 00:01:00 AM	0
00:00:00 AM – 00:02:00 AM	1
…	…
00:00:00 AM – 00:09:00 AM	8
00:00:00 AM – 00:10:00 AM	9
00:01:00 AM – 00:11:00 AM	10
00:02:00 AM – 00:12:00 AM	11
00:03:00 AM – 00:13:00 AM	12
00:04:00 AM – 00:14:00 AM	13
00:05:00 AM – 00:15:00 AM	14

Notice that the first nine window extents of Q5-1 are partial window extents, which do not have a full 10 minutes of tuples and only occur at the start point of the query.

The *extent*() function defines the content of each window extent. Given a window specification *S* and the set of tuples *T* in the input stream, *extent*() maps a window-id *w* to the subset *u* of tuples in *T* that belong to the window extent *w*: *extent*(*S*, *T*, *w*) = $U \subseteq T$. The *extent*() function can be naturally defined based on the meaning of the window. For example, in Q5-1, window 10 contains all the tuples with *ts* values where 00:01:00AM AM $\leq$ *ts* < 00:11:00 AM.

The *wids*() function indicates to which window extents a tuple belongs. The *wids*() function maps a tuple *t* to a subset *V* of window-ids in *W*: *wids*(*S*, *T*, *t*) = $V \subseteq W$. For example, a tuple *t* with *ts* value 00:00:05 AM belongs to windows 0 – 9, which can be derived based on *t.ts*: The first window-id for *t* is calculated by

(*t.ts* – start time) / SLIDE

= (00:00:05 AM – 00:00:00 AM) / 60 seconds

= 0.

The last window-id for *t* is calculated by

(*t.ts* + RANGE – start time) / SLIDE – 1

= (00:00:05 AM + 600 seconds – 00:00:00 AM) / 60 seconds – 1

= 9.

For a sliding-window query with a window specification RANGE 10 minutes and SLIDE 1 minute such as Q5-1, each input tuple belongs to a set of consecutive window extents. Tuple *t* above belongs to window extents 0 through 9—*wids*([10, 1, *ts*], *T*, *t*) = {0, 1, 2, …, 9}. Note that the *wids*() function does not require that each tuple belong to consecutive window extents, although that is true for most commonly used types of window. Also, in this example, the windows to which *t* belongs do not depend on *T*, though *T* is involved for some other kinds of window specifications. The *extent*() function and *wids*() function are duals of each other—the *extent*() function specifies the set of tuples in a window extent and the *wids*() function specifies the set of window extents to which a tuple belongs. The *extent*() and *wids*() functions define the logical window semantics from different perspectives. The *extent*() functions defines window semantics from a window-centric view—which tuples each window extent contains, while the *wids*() function defines it from a tuple-centric view—to which window extents each tuple belongs. We have found that the *extent*() function is typically more intuitive to define, and thus can be used as the reference to prove the correctness of its corresponding *wids*() function; the *wids*() function proves more useful in implementation, as we will discuss in Chapter 6.

The semantics of each type of window can be defined by providing these three functions. Next, we present the semantics of several common types of windows used by aggregation, by defining three functions for each window tuple.

5.1.2. Sliding Windows

For sliding-window aggregation, the sliding window separates the input stream into overlapping window extents, and an aggregate is computed over each window extent. The window specification for sliding-window aggregation consists of three parameters, RANGE, SLIDE and WA. RANGE specifies the length of the window; SLIDE specifies the step by which the window moves and thus how frequently an aggregate is computed; and WA is the windowing attribute—the attribute over which the window is specified. Potentially, WA can be any tuple attribute with an ordered domain, as long as it is a progressing attribute. We assume the arrival time and the arrival position of tuples in a stream are explicit attributes of the input tuples, called arrival-ts and row-num. Thus, either of these two attributes can serve as the WA attribute, in addition to any progressing attribute originally present. Q5-1 uses a sliding-window aggregation with window specification [RANGE 10 minutes, SLIDE

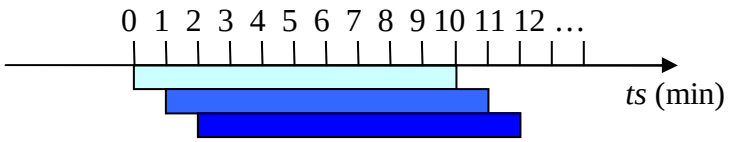


Figure 5-1 Three window extents of a sliding-window aggregation, Q5-1.

1 minute, WA *ts*]. It computes the number of packets from each source IP over each window extent (10-minute sub-stream). Figure 5-1 shows three consecutive window extents of Q5-1. Tumbling-window aggregation is a special case of sliding-window aggregation whose consecutive window extents do not overlap. For tumbling-window aggregation, RANGE equals SLIDE.

Following the WID semantics framework, we define the window semantics of sliding-window aggregation as follows. First, we use non-negative integers as window-ids. The *windows*() function is defined as below.

$$windows\ (T, S[RANGE\ r,\ SLIDE\ s,\ WA\ a]) = \{0, 1, 2, 3, \ldots\} \qquad \text{(Eq. 5.1)}$$

Next, using the defined window-ids to identify window extents, the *extent*() function defines the content of a window extent. That is, *extent*() maps a window-id, *w*, to a set of tuples in the window extent identified by *w*. The definition of *extent*() just follows the natural meaning of sliding-window aggregation. For ease of presentation, we assume that *RANGE*, *SLIDE* and *WA* attribute values are all in the same units.

$$extent(w, T, S[RANGE\ r,\ SLIDE\ s,\ WA\ a]) = \{t \in T \mid \min_a(T) + (w+1) * s - r \le t.a < \min_a(T) + (w+1) * s\} \qquad \text{(Eq. 5.2)}$$

In the definition of *extent*(), $min_a(T)$ represents the smallest value of the windowing attribute over all the tuples in *T*.

The *wids*() function maps an input tuple to a set of window extents to which the tuple belongs. It is the inverse of the *extent*() function. Let *W* = *windows* (*T*, *S*[*RANGE r*,

SLIDE s, WA a]). The *wids*() function for sliding-window aggregation is defined as follows:

$$wids(t,T,S[RANGE\ \ r,\ \ SLIDE\ \ s,\ \ WA\ \ a]) = \{w \in W \mid \lceil (t.a - \min_a(T))/s \rceil - 1 < w \leq \lceil (t.a + r - \min_a(T))/s \rceil - 1\}$$

(Eq. 5.3)

For example, the *wids*() function for Q5-1 is as follows:

$$wids(t,T,S[RANGE\ \ 600,\ \ SLIDE\ \ 60,\ \ WA\ \ ts]) = \{w \in W \mid \lceil (t.ts - \min_{ts}(T)/60) \rceil - 1 < w \leq \lceil (t.ts + 600 - \min_{ts}(T))/60 \rceil - 1\}$$

Suppose that the min value of the *ts* attribute of the input stream is 10:00:00AM, the window-ids of an input tuple *t* with *ts* value 10:00:05 AM can be calculated as follows. The first window-id for *t* is calculated as

$(\lceil (t.ts - min_{ts}(T)) /60 \rceil - 1) + 1$

$= \lceil$ (10:00:05 AM – 10:00:00 AM)/60 $\rceil - 1) + 1$

$= 0$

The last window-id for *t* is calculated by

$\lceil (t.ts + 600 - min_{ts}(T))/ 60 \rceil - 1$

$= \lceil$ (10:00:05 AM + 600 – 10:00:00 AM)/60 $\rceil$ / – 1

$= 9$

In sliding-window aggregation, as each tuple belongs to a consecutive set of window extents, tuple *t* belongs to window 0 through 9.

5.1.3. Partitioned Windows

A partitioned-window aggregate is similar to a sliding-window aggregate, but it uses an additional *partitioning attribute, PA*[2], to split the input stream into sub-streams (or partitions) before applying the other parameters in the window specification to each sub-stream. Q5-2, shown below, is a partitioned-window aggregate query; it is identical to Q5-3 except that the srcIP attribute in Q5-2 is a partitioning attribute instead of a group-by attribute.

Q5-2: "For each source IP, find the maximum packet length of the past 1000 packets from the source IP; update the results for every 10 packets from the source IP."

```
SELECT  srcIP, max(length)
            [RANGE 1000 rows, SLIDE 10 rows, WA row-num, PA srcIP]
FROM    Main
```

Q5-3: "For the past 1000 packets, find the maximum packet length from each source IP; update the result every 10 packets."

```
SELECT  srcIP, max(length)
            [RANGE 1000 rows, SLIDE 10 rows, WA row-num]
FROM    Main
GROUP-BY srcIP
```

However, the semantics of Q5-2 and Q5-3 are significantly different. Q5-3, a non-partitioned query, takes a sequence of 1000 tuples from the input stream as a window extent, then divides those 1000 tuples into groups by srcIP and counts the packets in each group. In short, Q5-3 first computes a window extent and then sub-divides that extent into groups. In contrast, Q5-2 first sub-divides a stream into "partitions" (sub-

[2] It is also possible that PA is a set of attributes.

streams) by the partitioning attribute, and then sub-divides each partition into window extents independently, based on the other three parameters in the window specification. The progress of each partition is independent of each other, and the number of window extents in each partition may differ. Note that for time-based window aggregates, the effect of a *PA* attribute is the same as using it as a group-by attribute [7], and thus for time-based partitioned-window, the *PA* parameter does not provide more expressive power.

The window semantics definition for row-based partitioned sliding-window aggregation is very similar to that of sliding-window aggregation, but it uses compound values, (*id*, *pa*), as window-ids—*id* is a non-negative integer representing the index of a window extent in the partition and *pa* is a partitioning-attribute value.

$$windows(T, S[RANGE\ r, SLIDE\ s, WA\ a, PA\ p]) = \{(id, pa) \mid id \in (0, 1, 2, \ldots), pa \in T.p\ \} \qquad \text{(Eq. 5.4)}$$

Here *T.p* means the projection of *T* on the partitioning attribute *p*.

The *extent*() function and *wids*() function for partitioned sliding-window aggregation are similar to Eq 5.2 and 5.3, respectively, but have an additional check on the partitioning attribute value of the tuple and the *pa* component of the window-id of the window extent to which the tuple belongs.

The *extent*() function in this case determines the content of the window extent based both on its integer index and partitioning attribute value. In the *extent*() function definition, we use the function *rank*(*t*, *attr*, *p*, *T*), which, given a tuple *t*, an attribute *attr*, a partitioning-attribute value *p*, and a set of tuples *T*, returns *t*'s rank in the *p*

partition of T, in the order of *attr*. For example, *rank*(*t*, *row-num*, *PA*, *T*) in the following extent function returns tuple *t*'s arrival position in the partition to which it belongs, i.e., *t.PA*.

$$extent\ ((id, pa), T, S[RANGE\ r, SLIDE\ s, WA\ row\text{-}num, PA\ p]) =$$
$$\{t \in T \mid t.p = pa,\ \min_{row\text{-}num}(T) + (id + 1) * s - r \leq rank(t.row\text{-}num, pa, T) < \min_{row\text{-}num}(T) + (id + 1) * s\} \quad \text{(Eq. 5.5)}$$

The *wids*() function is given below, where *rank* = *rank*(*t*, *row-num*, *pa*, *T*), and *W* is the set of window-ids defined by the *windows*() function in Eq. 5.4:

$$wids\ (t, T, S[RANGE\ r, SLIDE\ s, WA\ row\text{-}num, PA\ p]) =$$
$$\{(id, pa) \in W \mid t.p = pa, \lceil (rank - \min_{row\text{-}num}(T)) / s \rceil - 1 < id \leq \lceil (rank + r - \min_{row\text{-}num}(T)) / s \rceil - 1\} \quad \text{(Eq. 5.6)}$$

5.1.4. Landmark Windows

A landmark window is similar to a sliding window except that a tuple belongs to all window extents that begin after its arrival, and thus we use "ALL" as the RANGE parameter value in landmark-window specifications. Q5-4 below is a time-based landmark-window query, and it computes the number of packets coming from each source IP, and the SLIDE parameter indicates that the result will be extended in 1-minute increments—according to the *ts* attribute. It is similar to Q5-1, except that the scopes of the window extents of Q5-4 keep increasing, and each window extent subsumes all the previous ones.

Q5-4: "Count the number of packets from each source IP; update the results every minute."

SELECT srcIP, count(*) [RANGE ALL, SLIDE 1 minute, WA ts]

```
FROM  Main1
GROUP BY srcIP
```

The *windows*(), *extent*(), and *wids*() functions for landmark windows are defined as follows.

$$windows\ (T, S[RANGE\ ALL, SLIDE\ s, WA\ a]) = \{0, 1, 2, 3, \ldots\} \qquad \text{(Eq. 5.7)}$$

$$extent(w, T, S[RANGE\ ALL, SLIDE\ s, WA\ a]) = \{t \in T \mid t.a < \min_a(T) + (w+1)*s\} \qquad \text{(Eq. 5.8)}$$

$$wids(t, T, S[RANGE\ \ ALL,\ \ SLIDE\ \ s,\ \ WA\ \ a]) = \{w \in W \ \mid \ w > \lceil (t.a - \min_a(T))/s \rceil - 1\} \qquad \text{(Eq. 5.9)}$$

In the *wids*() definition in Eq. 5.9, W is the set of window-ids defined by the *windows*() function in Eq. 5.7.

5.1.5. Slide-by-Tuple Windows

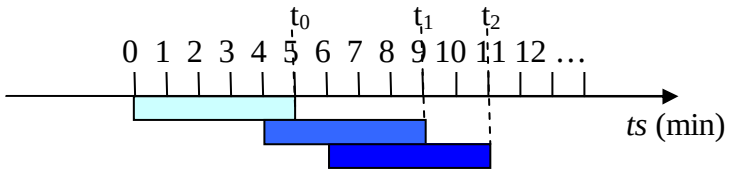


Figure 5-2 Three window extents of a slide-by-tuple window aggregation

A slide-by-tuple window is a special type of sliding window in which the RANGE and SLIDE parameter of a window are specified on different attributes. In such a case, SATTR (slide attribute) and RATTR (range attribute) are used in place of

WA to express the attributes over which SLIDE and RANGE are specified, respectively. A common example of this type of query is a query with RANGE over a timestamp attribute (RATTR) and a SLIDE of 1 row over *row-num* (SATTR). In such a case, each tuple arrival introduces a new window extent that has length RANGE and ends at the newly-arrived tuple. Query Q5-5 below is a slide-by-tuple window, and Figure 5-2 shows three window extents introduced by three tuples.

Q5-5: "Find the maximum packet length of packets for the past 5 minutes; update the result every tuple."

```
SELECT max(length)
        [RANGE 5 minutes, RATTR ts, SLIDE 1 row, SATTR row-num]
FROM Main
```

For this type of window, the number of window extents is data-dependent—a window extent is associated with each tuple. We do not use a simple integer sequence for window-ids; instead, we use values of *T.RATTR*—the projection of input tuples on *RATTR*—for window-ids. The *windows*() and *extent*() functions for slide-by-tuple windows are given below.

$$windows(T, S[RANGE\ r, RATTR\ ra, SLIDE\ 1, SATTR\ row\text{-}num]) = \{t.ra \mid t \in T\} \quad \text{(Eq. 5.10)}$$

$$extent(w, T, S[RANGE\ r, RATTR\ ra, SLIDE\ 1, SATTR\ row\text{-}num]) = \{u \in T \mid w - r < u.ra \leq w\} \quad \text{(Eq. 5.11)}$$

Assuming unique RATTR values, each RATTR attribute value identifies a distinct window extent that ends at that tuple. Let the set of window-ids defined by the *windows*() function in Eq. 5.10 be *W*. The *wids*() function for slide-by-tuple windows is given by:

$$wids\ (t, T, S[RANGE, r, RATTR\ ra, SLIDE\ 1, SATTR\ row\text{-}num]) = \{w \in W \mid t.ra \leq w < t.ra + r\} \quad \text{(Eq. 5.12)}$$

Here, the window-ids of window extents to which tuple t belongs fall between $t.ra$ and $(t.ra + r)$.

A more general form of the slide-by-tuple window has SLIDE as n tuples instead of one tuple. For example, the SLIDE parameter value of Q5-5 can be changed to 5 and then the window of Q5-5 advances every 5 tuples. Here, every n^{th} tuple defines a window extent. Thus, we use the *RATTR*-values of very nth tuples (i.e., n, $2n$, $3n$, …) in T as window-ids. The *windows*(), *extent*() and *wids*() functions of this type of window are given by:

$$windows(T, S[RANGE\ r, RATTR\ ra, SLIDE\ n, SATTR\ row\text{-}num]) = \{w \mid t \in T, mod(t.row\text{-}num, n) = 0,\ w = t.ra\} \quad \text{(Eq. 5.13)}$$

$$extent(w, T, S[RANGE\ r, RATTR\ ra, SLIDE\ s, SATTR\ row\text{-}num]) = \{u \in T \mid w - r < u.ra \leq w\} \quad \text{(Eq. 5.14)}$$

$$wids\ (t, T, S[RANGE\ r, RATTR\ ra, SLIDE\ s, SATTR\ row\text{-}num]) = \{w \in W \mid t.ra \leq w < t.ra + r\} \quad \text{(Eq. 5.15)}$$

The *extent*() and *wids*() functions in Eq. 5.14 and 5.15 are textually the same as those for the slide-by-tuple window in Eq. 5.11 and 5.12. But here in Eq. 5.15, W is the set of window-ids defined by the *windows*() function in Eq. 5.13. We assume that SLIDE of the slide-by-tuple windows is defined on tuples' arrival order to the stream system, which is what the *row-num* attribute indicates. To use tuples' arrival order to a specific window operator as SLIDE, we need the tuples' arrival order to that operator, which may not be the same as *row-num*. For example, if Select is used before the operator, some tuples may be filtered out. Also, tuples may become disordered during

processing before the window operator. However, knowing the tuples' arrival order at a window operator is not a big issue, as the order can be easily observed by the operator itself.

Another variation of the slide-by-tuple window, which is an even more general form, is where the SLIDE is *n* tuples over the logical order of the stream on the *SATTR* attribute. For example, the following query Q5-6 is such a query

Q5-6: "Count the number of packets for the past 5 minutes; update the result for every 5 tuples as defined by the *ts* attribute order."

```
SELECT count(*)
        [RANGE 5 minutes, RATTR ts, SLIDE 5 rows, SATTR rank(ts)]
FROM Main
```

The function *rank*(*ts*) maps each tuple *t* in the input stream to its rank in order of the *ts* attribute values. So instead of advancing a window based on tuple-arrival order, we advance it based on the logical order implied by *ts*. Thus, the window in Q5-6 is of length 300 seconds over the *ts* attribute, and slides by 5 rows over the logical order defined by *ts*. Conceptually, this window suggests sorting before windowing. Here, we only consider *rank*(RATTR)—the attribute defining the slide order needs to agree with the range attribute. The *windows*(), *extent*() and *wids*() functions of this type of window are defined below. The *windows*() function definition uses a *rank*(*t*, *attr*, *T*) function, which, given a tuple *t* and attribute *attr*, returns *t*'s rank in *T* in the order of *attr*. Here, we assume RATTR values are unique in the following function definitions. If the uniqueness of RATTR values is not guaranteed, we can use RATTR and the tuple arrival order together to determine a tuple's rank.

$$windows(T, S[RANGE\ r, RATTR\ ra, SLIDE\ n, rank(ra)]) = \{w \mid t \in T, mod(rank(t, ra, T)), n) = 0, w = t.ra\}. \quad \text{(Eq. 5.16)}$$

$$extent(w, T, S[RANGE\ r, RATTR\ ra, SLIDE\ n, rank(ra)]) = \{u \in T \mid w - r < u.ra \leq w\}. \quad \text{(Eq. 5.17)}$$

$$wids\ (t, T, S[RANGE\ r, RATTR\ ra, SLIDE\ n, rank(ra)]) = \{w \in W \mid t.ra \leq w < t.ra + r\}. \quad \text{(Eq. 5.18)}$$

In Eq. 5.18, *W* is the set of window-ids defined by the *windows*() function in Eq. 5.16.

Discussion: The window semantics definitions we present in this section cover almost all types of windows that we have seen in the literature. Plus, we believe the framework we present here for window semantics definition can be used for new types of windows, for example, windows with non-consecutive tuples or that overlap in a spatial domain. Further, the definitions of window semantics directly influence our implementation of window operators. As we will see in the next chapter, by introducing *wids*() and window-id into our implementation, which is called *WID*, our implementations do not need to assume ordered streams and are more efficient.

The complexities of The *extent*() and *wids*() functions are correlated and might affect the efficiency of our window aggregation implementations. The computation costs of the our window aggregation implementations are partly determined by how efficiently the *wids*() function can be evaluated, which is often inversely related to the complexity of the *wids*() function. As the *wids*() function is evaluated over each input tuple in our window aggregation implementations, a complex *wids*() function can increase the computation cost. The *wids*() functions we have given for existing types of windows are defined with linear expressions, and thus thye can be evaluated efficiently. Further,

the complexities of *extent*() and *wids*() functions determines whether it is possible to automatically derive the *wids*() function from the *extent*() function. As the *extent*() function is more intuitive to define, automatically deriving *wids*() function from the *extent*() function is convenient for users when introducing new types of windows. In general, without constraints on the operators that can be used in a function, inversing the function can be arbitrarily hard. For example, inverse functions may not exist for functions with *floor*(), *ceiling*(), *log*(), *exp*(), and high-order polynomial expressions. Functions with only plus, minus, multiplication, divide and low-order polynomial expressions can be inversed automatically.

5.2. Window-Join Semantics

Stream systems allow only joins whose state cannot grow indefinitely. The join operator in stream queries must have a condition on a progressing attribute of each input that ensures that every tuple can eventually be purged. This requirement indicates that a tuple *t* of one stream, L, should only join with a bounded range of tuples from the other stream, R. With the progress of the R stream, the tuple *t* can eventually be purged after it has been matched with all the potential R tuples with which it might be joined.

Tumbling-window join and sliding-window join are the most commonly used join operators in stream queries—windows are used to constrain the amount of state that join maintains so that the state does not grow without bound. Q5-7 and Q5-8 below are examples of tumbling-window join and sliding-window join, respectively.

Q5-7: “Find the network packet pairs from Main1 and Main2, in which the source IP of the Main1 packet matches the destination IP of the Main2 packet for each 5 minute interval.”

```
SELECT Main1.srcIP, Main1.destIP, Main2.srcIP, Main2.destIP, Main2.ts
FROM Main1, Main2
        [RANGE TUMBLING 5 minutes, WA ts],
WHERE Main1.srcIP = Main2.destIP
```

Q5-8: “Find the network packet pairs from Main1 and Main2 in which the source IP of the Main1 packet matches the destination IP of the Main2 packet; the Main1 packet should follow the Main2 packet within 2 minutes, and the Main2 packet should follow the Main1 within 3 minutes.”

```
SELECT Main1.srcIP, Main1.destIP, Main2.srcIP, Main2.destIP, Main2.ts
FROM Main1 [RANGE 3 minutes, WA ts],
         Main2 [RANGE 2 minutes, WA ts]
WHERE Main1.srcIP = Main2.destIP
```

It is intuitive to first think about “window join” assuming two input streams, M1 and M2, that are ordered, continuous, and synchronized. In such a scenario, we can consider window join as a join operator maintaining a tuple-buffer for each input stream to materialize a “window” of tuples; a tuple joins with all the tuples in the tuple-buffer of the other stream when it arrives. In other words, two tuples join if they have ever been in the tuple-buffers of the join at the same time. Tumbling-window join and sliding-window join differ in the way that they update the content of the tuple-buffers. For a tumbling-window join, when the tuple-buffer has a full window of tuples, the buffer is emptied and a new window starts. Note that for tumbling-window join, the sizes of the windows of the two inputs must be the same. For a sliding-window join, the tuple-buffer always maintains a full window of tuples (except at the very beginning of the query evaluation), and new tuples purge old tuples from the

buffer. For example, considering the tuple-buffer for the Main1 input of Q5-8, new tuples purge tuples that are more than three minutes older from the buffer.

Next, we present the semantics of tumbling-window join and sliding-window join without assuming physical arrival order of input streams. For ease of presentation, we sometimes assume the window condition is the only join condition and there are no other join predicates. In practice, there will be other join conditions, which can be viewed conceptually as a post-filter on the results of the join using just the window conditions.

5.2.1. Tumbling-Window Join and Sliding-Window Join

Just like window aggregation, window join is also defined on the progressing attributes of the input streams, and the window condition can be seen as an additional predicate of the join, regardless of the physical-arrival properties of input streams. The window condition of a tumbling-window join can be seen an equality predicate on (a function of) windowing attributes. Suppose that L and R are the left and right input, respectively. The window condition of a tumbling-window join is equivalent to an equality join predicate with integer division, $\lfloor L.A_L / W \rfloor = \lfloor R.A_R / W \rfloor$. Here, A_L and A_R are the windowing attributes of L and R and W is the window size. For example, in Q5-7, a 5-minute tumbling-window defined on an attribute *ts* of both input streams is equivalent to a equality join predicate $\lfloor Main1.ts/5 \rfloor = \lfloor Main2.ts/5 \rfloor$. The window condition of a sliding-window join can be seen as a band join predicate $L.A_L - W_L \le R.A_R < L.A_L + W_R$. Here W_L and W_R are the window sizes defined on L

and R, respectively. For example, Q5-8 is an example of a sliding-window join, defined on an attribute, *ts*, with 3-minute window on input L and 2-minute window on input R. A tuple, *l*, from L joins with a tuple, *r*, from R if $r.ts \geq (l.ts - 2 \text{ min})$ and $l.ts > (r.ts - 3 \text{ min})$ and if *l.srcIP = r.destIP*. Equivalently, this join specifies that a tuple, *r*, from R, joins with tuples with *ts* value greater than $(r.ts - 3 \text{ min})$ and smaller than $(r.ts + 2 \text{ min})$ from L when the IP addresses agree, which is band predicate $(r.ts - 3) < l.ts \leq (r.ts + 2)$. Seen this way, the semantics of both tumbling-window join and sliding-window join do not assume any physical stream properties, such as stream order or synchronization of the two input streams.

Discussion: In previous studies, window-join semantics have been blurred by confusion between stream progress and physical-stream-arrival properties. Algorithms proposed for implementing sliding-window join typically assume not only that each input stream of the join is ordered and continuous, but also assume that the arrival of input streams are synchronized—the *ts* value of an input tuple should be no smaller than the *ts* value of previously arrived tuples of either input. If the windowing attribute is arrival time, input streams for join naturally satisfy this "global order" property. Otherwise, using previously proposed algorithms requires maintaining global order, or incorrect results may be produced. For example, suppose the input stream Main1 for the sliding-window join query Q5-8 is delayed for 5 minutes and the join algorithm assumes "global order". When the tuple buffer for Main1 contains tuples with timestamps from 10:03:00AM through 10:05:59AM, the tuple buffer for Main2 will contain tuples with timestamp from 10:09:00AM through 10:10:59AM. Joining tuples

in the tuple buffers for Main1 and Main2, produces totally wrong results for Q5-8—for example, a Main1 tuple with timestamp 10:03:00AM and a Main2 tuple with timestamp 10:10:59AM will be joined.

5.2.2. An Alternative, Window-Semantic Definition for Window Join

We believe the window semantics for join can be defined in an alternative, window-oriented way. Just as in the WID semantics definition of window aggregation, each input stream can be separated into potentially overlapping window extents, which are represented by window-ids. Then, the join relationship can be defined on the window extents of each input stream of join—we define join between window extents based on window-ids.

In more detail, defining window semantics of windowed join in the window-oriented way has two parts, defining window extents on each input stream, and defining a join relationship between window extents. Then, two tuples join if they belong to window extents that can be joined. As a tuple may belong to multiple window extents, two tuples join as long as they have the sets of their window-ids overlap. Defining window extents for window join is the same as for window aggregation. To define the join relationship for the window extents requires a binary relation, *widjoin*, that contains pairs of matching window extents. Then, based on *widjoin* and the window condition of a join, we can derive a *match*() function that maps a tuple to a set of window extents with the content of which it should join: A tuple *t* is mapped to a window extent *w* if *t* belongs to some window extent *v* that matches *w* in the *widjoin* relation. Note that here

the *widjoin* relation defines window semantics for join from a window-centric view—which pairs of window extents should join. The *match*() function provides the same window-semantic information from a tuple-centric view—tuples of which window extents a tuple should join. We expect the *match*() function might be useful in the implementation of window join, just as the *wids*() function in the window semantic definition for aggregation. For a particular type of windowed join, we can define its window semantics by providing these required functions. Then, with the window-oriented semantics definition, the result of a windowed join is defined as the union of the result of joining each pair of window extents in *widjoin*. Here, L and R are the input streams of a windowed join; T_L and T_R are the set of tuples in L and R, respectively; $Spec_L$ and $Spec_R$ are the window specifications defined on L and R, respectively; p is the predicate of the join; and W_L and W_R are the window-ids for window extents defined for L and R, respectively.

$$result(T_L, Spec_L, T_R, Spec_R, widjoin(W_L, W_R), p) = \bigcup_{i,j \in widjoin(W_L, W_R)} extent(i, T_L, Spec_L) \bowtie_p extent(j, T_R, Spec_R) \quad \text{(Eq. 5.19)}$$

An example: In the following, we present the window-oriented semantics definition for sliding-window join, as an example to show how window join semantics can be defined in the window-oriented way.

We first define window extents on each input stream—we define them as advancing on every unit of the WA attribute values. Figure 5-3 shows window extents on the input streams, L and R, of a sliding-window join, with window specifications

[RANGE 3 minutes, WA ts_R] and [RANGE 3 minutes, WA ts_S] on L and R, respectively. Suppose the unit of the *ts* attributes is seconds, then the window extents defined on L and R are the same as window extents for a sliding-window aggregation with window specification [RANGE 180 seconds, SLIDE 1 second, WA ts_R] and [RANGE 180 seconds, SLIDE 1 second, WA ts_S], respectively—each window extent is a 180-second sub-stream and consecutive window extents overlap by 179 seconds. Assuming R and S start at the same time (i.e., L and R has the same min(*ts*) value), then a window extent *w* on L joins with the window extent on R with the same window-id. Thus, let W_L and W_R be the set of window extents defined on L and R, respectively, the *widjoin* relation is defined as follows.

$$widjoin(W_L, W_R) = \{(i, i) \mid i \in W_L,\ i \in W_R\} \qquad \text{(Eq. 5.20)}$$

If L and R do not start with the same *ts* value, we can use the smaller *ts* value as the start *ts* value for both L and R. This way, the input stream with the larger start *ts* value has empty window extents defined, but the *widjoin* relation remains the same.

A tuple from each input stream joins with tuples of a set of window extents on the other input stream. Here UNIT is the unit for WA values (or the UNIT of the one with finer granularity) and we assume the WA values start from 0. The *match*() functions for L and R are defined as follows. Basically, each tuple joins with every window extent on the other side with window-id in the range of the window-ids of the tuple itself. Note that the *match*() functions uses a band condition on the window-ids of window extents.

$$match_L\ (l, T_L, S_L\,[RANGE_L, WA_L], T_R, S_R\,[RANGE_R, WA_R]) =$$

$$\{w \in W_R \mid l.WA_L - 1 < w \leq \lceil (l.WA_L + RANGE_L)/UNIT \rceil - 1\}$$

(Eq. 5.21)

$$match_R\,(r, T_R, S_R\,[RANGE_R, WA_R], T_L, S_L\,[RANGE_L, WA_L]) =$$

$$\{w \in W_L \mid r.WA_R - 1 < w \leq \lceil (R.WA_R + RANGE_R)/UNIT \rceil - 1\}$$

(Eq. 5.22)

Here, l and r are tuples from L and R, respectively; W_L and W_R be the set of window extents defined on L and R, respectively; T_L and T_R are tuples in L and R, respectively; S_L and S_R are window specifications defined on L and R, respectively. Also, we assume that WA_L, WA_R and UNIT are the same granularity; or coarser units are converted to finer.

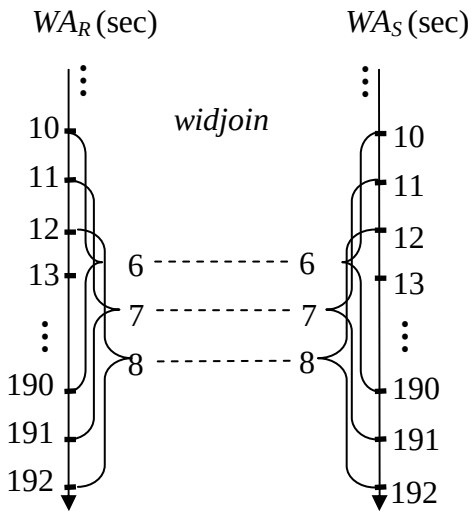


Figure 5-3 The widjoin relation for a sliding-window join—a window extent k of R joins with the window extent k of S.

In summary, we have discussed that the window condition for the most commonly used two types of window join, tumbling-window and sliding-window join, can be

clearly expressed with equality or band predicates on the windowing attributes. We believe that the window-oriented way of defining window join semantics is generic and expressive and can potentially be used to define semantics for any type of window join. With the window-oriented approach, window semantics of different types of windows can be defined in the same framework, and thus the semantics of different types of window join can be compared to each other.

Chapter 6

ORDER-INSENSITIVE IMPLEMENTATIONS OF WINDOW AGGREGATION

In this chapter, we present order-insensitive implementations of window aggregation. Order-insensitive implementations of query operators process tuples on the fly without requiring or enforcing order on the input. Instead of relying on stream order to determine the boundaries of window extents, these implementations leverage *punctuation* to communicate the completion of extents. In this chapter, we assume the granularity of punctuation is the same as the granularity of the window slide parameter. We discuss punctuation generation in Chatper 8. Three implementation algorithms for window aggregation are proposed, *WID*, *Paned-WID* and *AdaptWID*. WID is an implementation based directly on the WID window semantics described in Chapter 5. We categorize different types of windows used by aggregation based on the information that each type of window requires in order to map tuples to window-ids. This categorization distinguishes different requirements in the WID implementation for different categories of windows. Paned-WID extends WID with shared sub-aggregation to reduce computation cost. AdaptWID combines the WID implementation and the buffering implementation to reduce the memory cost of aggregation when the input data distribution is skewed. All three algorithms are order-insensitive implementations and assume the presence of punctuation to notify query operators about the ends of extents.

6.1. The WID Implementation

The WID implementation is a direct application of our window semantics, and of the *wids*() function in particular. The WID implementation uses window-ids to encapsulate window semantics. Further, WID explicitly transforms the window semantics of queries into data semantics via a *wid* attribute. In short, WID tags each input tuple with window-ids using the appropriate *wids*() function, and then uses the window-ids as an additional grouping attribute for the aggregate operator. In more detail, WID introduces a new operator, *Bucket*, that implements the *wids*() function and tags each tuple with its window-ids. The window-ids are appended to tuples as an explicit data attribute *wid*. Aggregate operators include the *wid* attribute with the grouping attributes defined in the query, and compute the aggregate value for all groups defined by the combined set of grouping attributes. The ends of window extents are signaled by punctuations. For example, suppose the timestamp values of the input stream of the query Q6-1 shown below start at 12:00:00. When a punctuation <*, *, *, *, *, 12:11:00> arrives, it indicates that all the packets with *ts* value smaller than 12:11:00 have already arrived and thus window 10 is complete as are previous window extents. Here, the *ts* attribute is called the *punctuating attribute*. Typically, the punctuating attribute is the progressing attribute of a stream. In a window query, the windowing attribute should be the progressing attribute of the input stream, and thus the punctuating attribute is also the windowing attribute.

Q6-1: "Count the number of packets from each source IP from the past 10 minutes; update the results every minute."

```
Select srcIP, count(*)  [RANGE 10 minutes, SLIDE 1 minute, WA ts]
From Main
Group By srcIP
```

Readers may wonder how the result of window aggregation should be interpreted, as the result of our window aggregation implementation is not ordered and does not have a timestamp attribute associated with it. The result of window aggregation represents aggregate values over time ranges. Thus, in general we cannot append a single timestamp value to the window aggregation result. (Tumbling-window aggregation is special because the time range can be represented as a simple function of timestamps.) However, a progressing attribute is needed for the output of window aggregation so that down-stream operators can progress. In our WID implementations, the output stream of window aggregation has an implicit *wid* attribute as the progressing attribute. Also, remind that the *extent*() function maps a window-id back to the set of tuples in the window by a condition on the timsestamp value of tuples. We could implement a similar function to map a *wid* attribute value back to the timestamp range of the window extent when presenting results to users.

6.1.1. An Example

Figure 6-1 shows a query plan for evaluating the sliding-window query Q6-1 using WID. The query plan consists of two query operators, the Bucket operator, which tags input tuples with window-ids using the wid attribute, and the aggregate operator, Count, which uses the wid attribute as a grouping attribute to compute window aggregation. The wid attribute contains a range value that indicates the range of

window-ids associated with each tuple. For example, t1 belongs to windows 10 to 19, and thus the Bucket operator appends a wid attribute value, 10–20, to t1 and outputs a

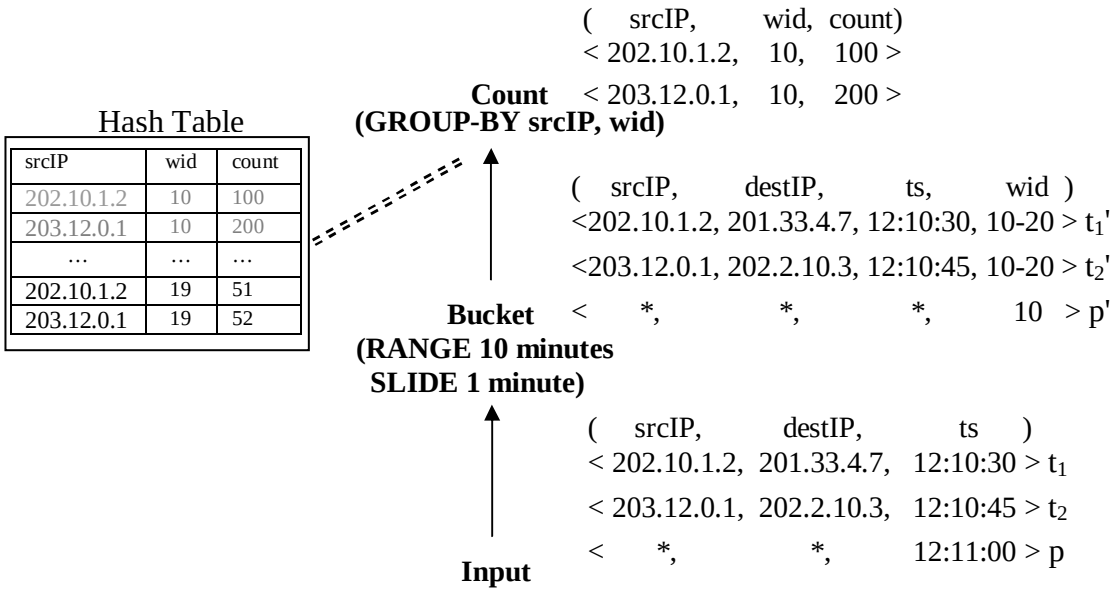


Figure 6-1 A query plan for Q6-1 using WID

tuple, t_1'. Here, 10–20 represents the interval [10, 20). The Count operator groups on he *srcIP* and *wid* attributes, and incrementally maintains the count of packets for each group in a hash table structure. It uses each tuple to update the groups within the tuple's *wid* range. For example, the tuple t_1' is used to update 10 groups, for windows 10 to 19. Note that here the Bucket operator could replicate a tuple 10 times and tag the tuple copies individually with a window-id for each. Then the Count operator would be a normal punctuation-aware aggregate operator, and need not handle range values. We use range values for the *wid* attribute to avoid increasing the data volume of the inter-operator stream between the Bucket and Count operator, at the cost of slightly more complexity in the implementation of Count. In Figure 6-1, the ends of

window extents are marked by punctuations. For example, punctuation *p* indicates that all the tuples with *ts* value smaller than 12:11:00 have arrived, and is translated by the Bucket operator into a punctuation *p*' that indicates the end of window 10. The punctuation *p*' unblocks the Count operator—it allows the Count operator to output the aggregates that match *p*'.

The WID implementation provides one-pass query evaluation for sliding-window aggregate queries, eliminating the need to materialize window extents (i.e., retain input tuples in an intra-operator buffer), and thus can greatly reduce memory usage during query evaluation. The WID implementation is very flexible and scalable. It does not put constraints on physical properties of the input streams such as arrival order and continuity. Some other window aggregate implementations, such as the buffering implementation, require the data be sorted before being aggregated. In contrast, WID does not have such constraints. In addition, the aggregation step is window-agnostic since *wid* is treated as a data attribute, and the implementation of the window semantics is easy to manage and verify, as it is isolated in the Bucket operator.

The detailed WID implementation varies for different types of windows. Before going into the details, we first introduce the concept of context and present a categorization of windows based on the "context" that different types of window aggregation require in order to map tuples to window-ids (i.e., to implement the *wids*() function in the Bucket operator). Categorization helps to determine the appropriate implementation techniques for given types of windows.

6.1.2. Categorization of Windows

We define two types of "context" information that may be used in mapping tuples to window-ids: *backward-context* and *forward-context*. For a tuple *t*, its backward-context is information about tuples that have arrived before *t*. Forward-context is information about tuples that will arrive after *t*. If a *wids*() function requires backward-context, it implies that the implementation will need to maintain information about previously arrived tuples. For example, the implementation of a partitioned tuple-based window must maintain a count of tuples that have arrived for each partition. The *rank*() function in the *wids*() definition for tuple-based partitioned windows reflects a backward-context requirement, because *rank*() needs to return a tuple's rank in the partition it belongs to and thus requires knowledge of the number of tuples in the partition ranked before the tuple. Typically, having to maintain backward-context is not a significant restriction, because it does not prevent one from determining window-ids for tuples on the fly. However, if a *wids*() function requires forward-context, that means that information from tuples arriving after a tuple *t* is required to calculate the window-ids for *t*. This requirement implies that the exact window-ids for tuple *t* cannot all be determined until those tuples arrive. Thus a *wids*() function requiring forward-context implies that tuples may need to be buffered and delayed. Slide-by-tuple windows require forward-context. The use of the WA values of later tuples (i.e., $t.RATTR \leq w < t.RATTR + RANGE$) in the *wids*() definition for slide-by-tuple windows reflects a forward-context requirement.

Based on their forward-context requirements, we categorize windows into *FCF* (*f*orward-*c*ontext *f*ree), and *FCA* (*f*orward-*c*ontext *a*ware). We define a window as FCF if the *wids*() function does not require forward-context and thus the set of window-ids for each tuple can be determined on the fly. Time-based windows, tuple-based sliding windows and partitioned windows are FCF. We define a window as FCA (*f*orward-*c*ontext *a*ware) if the *wids*() implementation requires forward-context and thus the set of window-ids for each tuple cannot be determined on the fly. Slide-by-tuple windows and its two variations (slide by *n* tuples over row-num and *rank*(*RATTR*)) are FCA.

Within the FCF category, we define a window as *CF* (*c*ontext *f*ree) if the implementation of its *wids*() mapping requires neither forward- nor backward-context. Tuple-based and time-based sliding windows are CF. The *wids*() function of a CF window maps each input tuple to a set of window-ids based only on the window specification and the tuple itself, and correspondingly in the implementation, window-ids for each tuple can be determined as the tuple arrives and no state needs to be maintained. Next, we discuss the implementation details of the Bucket and Aggregate operators for commonly used types of windows.

6.1.3. The WID Implementation for FCF Windows

For FCF windows, the Bucket operator tags each tuple with a window-id range, which represents the set of window-ids in the range. The Aggregate operator is window-agnostic—it uses the *wid* attribute as an additional grouping attribute. (Although the *wid* attribute contains range values, an Aggregate operator might support such range

values, as well as overlapping groups, for purposes other than windows. For example, the Aggregate operator for querying spatial data may also need to support range values and overlapping groups.) Next, we discuss the WID implementation for FCF windows, which consists of two operators, the Bucket operator and the Aggregate operator.

6.1.3.1. Bucket

The first step in the WID implementation is to tag each tuple explicitly with window-ids. The Bucket operator takes a window specification as a parameter, and tags each tuple with its associated window-ids by using the appropriate *wids*() function. The basic structure of the Bucket implementation is straightforward as shown in Figure 6-2. The *Bucket*() function is called for each input item. We use a range-value attribute to represent the range of window-ids for each tuple. The *processTuple*() function calls the *wid_bounds*() function to get a pair of values, *wid_start* and *wid_end*, which it appends to the input tuple. The *wid_bounds*() function computes *wid_start* and *wid_end* based on the *wids*() function defined for the type of window. The *wid_start* value indicates the first window extent to which the tuple belongs; and the *wid_end* value indicates the first window extent to which the tuple does not belong. Here we

```
State Maintained:
range:dow size of the aggregation;
slide: window slide of the aggregation;
wa: windowing attribute used;

Bucket(x)
if x is a tuple
   ProcessTuple(x);
else if x is a punctuation
   ProcessPunctuation(x);
ProcessTuple(t)
(wid-start, wid-end) = wid_bounds(t);
create t' by appending the range value, (wid-start, wid-end), to t as the wid
attribute;
output t';
ProcessPunctuation(p)
(wid-start, wid-end) = wids_bounds(p);
create p' by appending wid-start to p;
change the wa value of p' to *;
output p';
wid_bounds(t)
wid-start = lower bound of wids([range, slide, wa], t.wa);
wid-end = upper bound of wids([range, slide, wa], t.wa);
return (wid-start, wid-end);
```

Figure 6-2 Order-insensitive implementation of window aggregation: the Bucket operator

assume a tuple belongs to a consecutive set of window extents. In the rest of the discussion, we use the phrase "the range of window-ids" to refer to this pair of values. Punctuation on the windowing attribute is turned into punctuation on the *wid* attribute. The *processPunctuation*() function applies the same *wid_bounds*() function to punctuation and appends the *wid_start* value computed as the *wid* attribute value for the punctuation. In addition, the windowing attribute value of the punctuation is turned into a wild card (indicating this attribute can match any value). Here we assume linear

punctuation on the windowing attribute. Note that all the complexity of tagging tuples with window-ids is encapsulated in the Bucket operator. Figure 6-3 shows the query plan using the WID implementation for Q6-1, which is a CF query.

A key difference in the Bucket operator for various types of windows is the amount of tuple state that the Bucket operator must maintain. For CF windows, Bucket does not need to maintain any tuple state and can append a range of window-ids to each input tuple immediately when the tuple arrives, since the *wids*() function for a CF window requires no context information. For windows that are FCF but not CF, Bucket may need to maintain state for previously arrived tuples. For example, for tuple-based partitioned windows, the Bucket operator needs to remember the count of tuples that have arrived for each partition and then the window-ids tagged for each tuple are computed by using the count when the tuple arrives as the windowing attribute value.

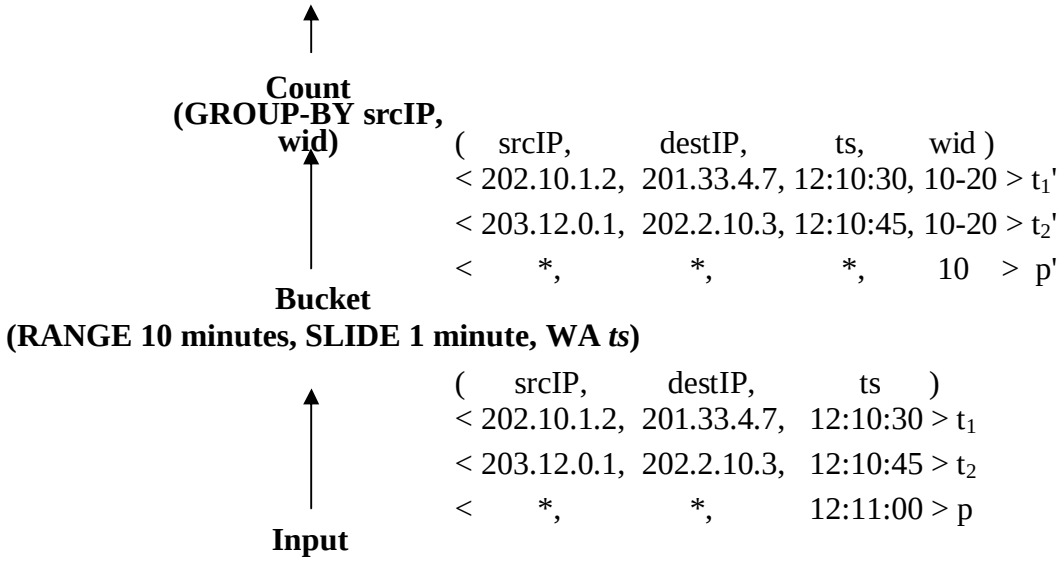


Figure 6-3 Query plan for Q6-1 with the WID implementation

6.1.3.2. Aggregation

Given a tuple *t* tagged with a range of window-ids, (*wid_start, wid_end*), an Aggregate operator, such as Max, uses *t* to update the n aggregate values whose *wid* values fall between *wid_start* and *wid_end*. We have extended the Aggregate operator to understand range values. The implementation for the Aggregate operator is shown in Figure 6-4. In this implementation, the *Aggregate*() function is called for each tuple. Aggregates for window extents are incrementally updated with tuples in the extent using the *ProcessTuple*() function and a hash table is used to maintain these aggregates—how exactly the *ProcessTuple*() function updates the aggregates depends on the aggregate function being computed. Note that with explicit window-ids, the window specification and thus the window semantics is not exposed to the Aggregate operator. When punctuation arrives, the hash-table maintained by the Aggregate operator needs to be scanned in order to output the appropriate aggregate values. An alternative that avoids a hash-table scan is to output aggregates on hash-table collisions, similar to the slow flush mechanism to be discussed in Chapter 8. In contrast to implementations that hardwire arrival-order assumptions into the implementation, using punctuation to signal the ends of window extents is more flexible.

```
State Maintained:
ht: hashtable maintaining partial window aggregates;
gpattr: the grouping attributes of the aggregation;

Aggregate(x)
if x is a tuple
   ProcessTuple(x);
else if x is a punctuation
   ProcessPunctuation(x);

ProcessTuple(t)
for each wid in [t.wid-start, t.wid-end)
   compute hash value, hval, for t with t.gpattr and wid;
   update the aggregate value maintained in ht[hval] using t;

ProcessPunctuation(p)
scan ht and output any group with wid value equaling p.wid-start;
output a punctuation with value p.wid-start;
```

Figure 6-4 Order-insensitive implementation of window aggregation: the Aggregate operator

For tuple-based window aggregation, WID assumes an explicit tuple sequence number, *seq-num*. Thus, if a count-based window is defined on tuple-arrival order (arrival at the stream system, not the Aggregate operator), the stream system needs to tag each input tuple explicitly with a sequence number representing the tuple's arrival order, and insert punctuations on the *seq-num* attribute of the input tuples. Then, the Aggregate operator can use *seq-num* as the windowing attribute. However, if the window is defined on the tuples' arrival order at the Aggregate operator, WID itself needs to tag each tuple with its *seq-num*. For tuple-based sliding-window aggregation, the Bucket operator needs to maintain the count of tuples that have arrived, tag each tuple with its *seq-num*, and also insert punctuation when a window extent ends. When

the Aggregate operator receives a punctuation, it first outputs results for the ending extent and then produces a punctuation for it. For tuple-based partitioned-window aggregation, the Bucket operator needs to maintain the count of tuples for each partition, and tag each tuple with its rank in its partition; also the Bucket operator needs to insert punctuation for each partition. Thus, the Aggregate operator for tuple-based partitioned-windows receives punctuation on both window-id and the partitioning attributes. Therefore, the Aggregate operator outputs results individually for each partition in a window extent, and also produces punctuation for each partition in the extent.

The correctness of punctuations affects the accuracy of results. We assume punctuation is "grammatical" in this thesis. The regular arrival of punctuations can reduce the delay in outputting results. Delays in punctuation arrival delay results, and increase the state that the Aggregate operator must keep, but do not affect the correctness of results.

6.1.3.3. Summary and Discussion

In addition to naturally accommodating out-of-order tuples, WID is also more flexible in implementation. WID decomposes window-aggregate evaluation into several parts, including implementation of window semantics, detection and notification of the ends of window extents, and internal management of the state required for aggregation. Compared to the buffering implementation, this decomposition allows each part to be more independent of the others and thus allows a more flexible implementation

overall. In WID, the window semantics is implemented explicitly by the Bucket operator and is encapsulated in that operator. This encapsulation allows other parts of the implementation (e.g., state management for aggregation) to be window-agnostic. Therefore, WID can support different types of windows easily—the only part of the implementation that may vary with different types of windows is the computation of window-ids in the Bucket operator. The ends of window extents are signaled by punctuations. The Aggregate operator implementation resembles that of the relational aggregate operator, although it outputs results incrementally. In our current implementation, the Aggregate operator maintains partial aggregates for each group using a hash table. However, the internal state that the Aggregate operator maintains and the data structure used for the internal state are decisions local to the Aggregate operator and are independent of the other parts of the implementation. For example, in the buffering implementation, the content of a window extent is associated with the tuples buffered by the Aggregate operator, while in WID, the tuples are tagged by the Bucket operator with the window extents to which they belong, independent of the implementation of the Aggregate operator.

In terms of performance, we believe WID has several advantages over buffering, including reducing memory usage, latency, and execution time. These improvements are discussed further below.

Reducing memory usage: WID reduces memory usage by avoiding buffering input tuples. The memory requirement of the Bucket operator is minimal and the Aggregate operator maintains only one aggregate value for each group in each open extent. The

main space savings come from never explicitly materializing window extents, but instead incrementally maintaining aggregates for multiple window extents simultaneously—almost always a beneficial tradeoff. For example, if RANGE is 60 minutes, and SLIDE is 5 minutes, current window-query evaluation algorithms would buffer one hour's worth of tuples; in contrast, the WID approach needs to buffer only 12 (= 60/5) aggregate values—one for each active window extent. Secondary space savings come from avoiding any buffer space devoted to sorting out-of-order tuples. The tuples can be tagged and processed as they arrive. The only offsetting expense is sometimes retaining a few more aggregate values for incomplete window extents.

Reducing latency: WID incrementally maintains window aggregates, and thus avoids the response delay that the basic buffering implementation requires due to scanning and aggregating tuples at the end of a window extent. WID can output the results for the window extent immediately upon the arrival of the punctuation covering the extent. (A punctuation covers a window extent if the range of the extent is within the range of the punctuation; for example, in Q6-1, the punctuation <*, *, 10:20 AM> covers the window extent [10:10 AM, 10:20 AM).) When the input stream contains delayed tuples, WID may have even more latency advantages, because punctuation can express end-of-extent messages promptly, while the mechanisms that buffering uses to deal with disorder, such as heartbeats or slack, must provide for the worst-case disorder to guarantee the accuracy of the results. However, we also note that the latency of the buffering implementation can be improved by both buffering tuples and maintaining aggregates, at the cost of extra memory usage for maintained aggregates.

Reducing execution time: WID potentially uses less CPU time than the buffering technique. As the window semantics information is tagged onto each input tuple, WID handles each tuple only once in order to update all the partial aggregates of window extents to which that tuple belongs. Recall that in the buffering implementation, each tuple is stored in the buffer and revisited multiple times—once for each window extent to which the tuple belongs. We note that for certain aggregates, such as Count and Sum, the execution time of the buffering implementation can be reduced by leveraging the aggregate of the previous extent to compute the aggregate of a window extent. For example, to compute window count, the count for a window extent can be initialized by the count for the previous window extent minus the number of expired tuples and thus the cost of re-scanning the unexpired tuples can be avoided.

6.1.4. The WID Implementation for FCA Windows

For FCA windows, we cannot calculate the set of window-ids for a tuple *t* on-the-fly, since this would require information about tuples arriving in the future. In many cases, the requirement of forward-context leads to buffering and delaying tuples. However, careful examination of the *wids*() function for slide-by-tuple windows and two of its generalized forms reveals that we can determine on the fly for each tuple the range into which these window-ids will fall, but not the exact set of window-ids. For example, given the range of a slide-by-tuple window, RANGE, and a tuple *t* with *t.RATTR* = *s*, the set of windows-ids to which *t* is mapped fall into the range [*t.RATTR*, *t.RATTR* + RANGE), and thus Bucket will tag *t* with this range. (Recall that for slide-

by-tuple windows and variations, we use the values of the attribute on which the range parameter is defined, *RATTR*, as window-ids; also, because of that, the range of the window-ids of a tuple does not determine the set of window-ids for it.) This range has a different meaning from that used for FCF windows, and the binding of window-ids to input tuples has to be deferred to the Aggregate operator.

Below, we present a one-pass algorithm for the Aggregate operator for slide-by-tuple windows with time-based ranges. This algorithm processes each tuple only once and handles out-of-order tuples the same as in-order tuples. Basically, we avoid retaining and re-processing tuples by maintaining partial aggregates for extents and by using these partial aggregates to initialize partial aggregates for new extents.

6.1.4.1. Slide-by-tuple windows

We start with an example first. Remember that we use tuples' windowing-attribute values as window-ids for slide-by-tuple windows. Each tuple starts a new window extent that ends with the tuple, and we use the tuple's window attribute for the window-id of the extent. Thus, for each input tuple t with $t.RATTR = s$, the first window extent t belongs to has window-id s. Further, $extent(s) = \{u \in T \mid s - RANGE < u.RATTR \leq s\}$, which ends when all tuples with *RATTR* value no more than the *RATTR* value of t have arrived. We also define an auxiliary extent for t that is the earliest subsequent extent to which t does not contribute—$aux\_extent(s) = \{u \in T \mid s < u.RATTR \leq (s + RANGE)\}$. Note that $aux\_extent(s) = extent(s + RANGE)$. Here, $extent(s + RANGE)$ does not necessarily correspond to a tuple in T—there might not

be a tuple with windowing attribute equal to *s* + *RANGE*. For ease of presentation, we denote the window extent and the auxiliary extent of tuple *t* with *RATTR* value s as Ss and Es respectively, and refer to them as *bins* collectively. One can think of Ss and Es as the "start bin" and "end bin" for *t*, respectively; and Ss has bin-id *s* and Es has bin-id (*s* + *RANGE*).

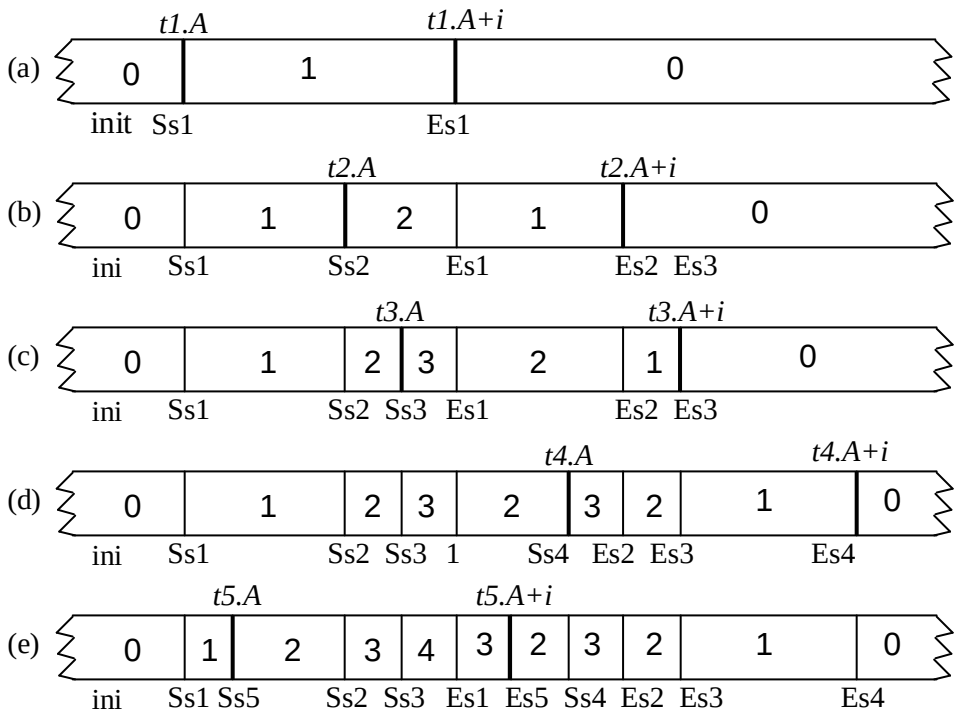


Figure 6-5 Example of insertion, initialization, and update of bins as new tuples arrive for slide-by-tuple count

Figure 6-5 shows the processing of a slide-by-tuple query where the aggregate is count, the *RATTR* is *A*, and RANGE is *i*. We depict the bins as laid out in order of the *A* attribute. Let *s*j = *t*j.*A*. We use Ssj and Esj to refer to the start bin and the end of tuple *t*j, and, in Figure 6-5, a bin-id is associated with the end of each bin. We mark

the region from the end of a bin, *b*, to the end of the next bin with the partial aggregate value for the bin *b*. For example, in Figure 6-5(d), the partial aggregate for Es1 is 2 and for Ss4 is 3. The reason that we label regions in this way is to indicate that any later bin whose bin-id is in the region would have that contribution to its partial aggregate from tuples seen so far. For example, as shown in Figure 6-5(e), the start bin of *t*5, Ss5, initiated with the arrival of *t*5, has a contribution of 1 to its count from tuples arrived so far. Futher, the partial aggregates of bins are updated incrementally—the partial aggregates of bins between the start bin Ss5 and the end bin Es5 of t_5 are incremented by 1 with the arrival of t_5.

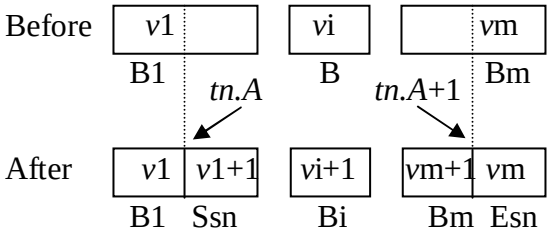


Figure 6-6 Bin updates for arrival of tuple tn

Let us examine the stages in Figure 6-5 sequentially, and consider the arrival of tuples $t_1 - t_5$. We start with an initial special bin, init, with count = 0. The arrival of t_1 adds bins Ss1 and Es1 (Figure 6-5(a)), with initial values 1 and 0, respectively. Tuple t_2 with $s_2 > s_1$ starts bins Ss2 and Es2, with Ss2 set initially to the value of Ss1 incremented by 1 (because Ss2 has the contribution from both t_1 and t_2), and Es2 initialized to Es1 (Figure 6-5(b)). Es1 is incremented by 1, to reflect the contribution of t_2. Figure 6-5(c) shows the effect of t_3, where $s_3 > s_2$: Ss3 and Es3 are created and

initialized, and Es1 and Es2 are incremented. Figure 6-5(d) shows the need for E-bins: Ss4 is initialized from Es1, reflecting the contribution of t_2 and t_3, but with t_1 out of the extent for Ss4. Finally, Figure 6-5(e) shows the arrival of an out-of-order tuple t_5, with $s_1 < s_5 < s_2$. Ss5 is initialized from Ss1 and Es5 from Es1, with bins Ss2, Ss3 and Es1 incremented. If at this point, punctuation arrives indicating future *A* values are greater than s_2, the operator can emit the aggregate values for Ss1, Ss5 and Ss2—the start bin of t_1, t_5, and t_2, and discard Ss1 and Ss5.

Figure 6-6 shows the general case for the arrival of tuple t_n, when (Ssn, Esn) spans bins B1, B2, …, Bm. Bins B1 and Bm are "split" and used to initialize Ssn and Esn; every bin Bi, $1 < i \leq m$ is also updated to reflect the contribution of t_n.

Figure 6-7 contains the algorithm for the aggregate operator for slide-by-tuple windows. The aggregate operator needs to store partial aggregates for bins that are not expired. Initialize sets up the special "*init*" bin, labeled with -∞. The *ProcessTuple*() function sets up new start and end bins for each arriving tuple, then updates the appropriate intervening bins. The *ProcessPunctuation*() function outputs results and purges the appropriate bins. This algorithm for slide-by-tuple windows avoids reprocessing tuples at the cost of maintaining auxiliary extents (end bins); but, on the other hand, it does not need space to retain input tuples. Also, it maintains partial aggregates for active window extents incrementally. Therefore, we expect that this algorithm will compare favorably to the buffering implementation in terms of execution-time performance and latency performance, and will be comparable in terms

*St*two collections, S and E, each storing pairs of the form [*bid*, *pa*] where *pa* is the partial aggregate for bin with bin-id *bid*. S stores start bins and E stores end bins.

Initialize()
/* *aggr-init* depends on the aggregate function; for example, *aggr-init* = 0 for count */
/* We use -∞ as the bin-id of the init bin*/
add [-∞, *aggr-init*] to E

Aggregate(*x*)
if *x* is a tuple
ProcessTuple(*x*);
else if *x* is a punctuation
ProcessPunctuation(*x*);

ProcessTuple(*t*)
/* Let the bin-ids of start-bin and end-bin of t be Ss and Es. */
t.wid=(Ss, Es)
add [Ss, *pa*] to S, where [*w*, *pa*] ∈ S ∪ E has the largest bin-id *w* < Ss
add [Es, *pa*] to S, where [*w*, *pa*] ∈ S ∪ E has the largest bin-id *w* < Es
/* the update operation depends on the aggregate-function; for example, if aggregate-function = count, the update operation is +1 */
for each [*w*, *pa*] in S ∪ E where Ss ≤ *w* < Es
update *pa* using *t*

ProcessPunctuation(*p*)
Output each [*w*, *pa*] in S with *w* < *p.wid* and remove it from S
Remove each [*w*, *pa*] in E with *w* < *p.wid* and *w* ≠ -∞

Figure 6-7 The Aggregate operator implementation for slide-by-tuple windows

of memory usage. However, implementation and testing of this variant remains as future work.

6.1.4.2. Variations

The algorithm for slide-by-tuple windows in Figure 6-7 can be extended to support the two variations of slide-by-tuple windows discussed in Section 5.1.2, again with the cost of maintaining partial aggregates for additional extents. No tuples need to be retained and reprocessed. The Bucket operator for these two variations is the same as for slide-by-tuple windows. For the variation that slides over the *seq-num* attribute, the *ProcessTuple*() function in the aggregate operator still maintains partial aggregates for two bins, Ss and Es for each tuple *t*; but it stores the *t.seq-num* with the two partial aggregates for it, e.g., [Ss, *t.seq-num, pa*]. The *ProcessPunctuation*() function only outputs the aggregates for the required window extents. For example, if the *SATTR* parameter is *seq-num* and the SLIDE parameter is 3 tuples, only aggregates with *seq-num* as a multiple of 3 are output. Similarly, for the variation that slides over the tuple count of the logically-ordered input stream over *RATTR*, the *ProcessTuple*() function stores the current rank (based on *RATTR*) of *t* with the partial aggregates, e.g., [Ss, *tup-rank, pa*]. The stored tuple rank may be updated as a new tuple arrives—for example, if a tuple *s* is delayed, its arrival will cause the increment of *tup-rank* of bins for tuples with *RATTR* value greater than *s.RATTR*. The *ProcessPunctuation*() function only outputs the aggregates for the required window extents. For example, if the *SATTR* parameter is *tup-rank* and the SLIDE parameter is 3 tuples, only aggregates with *tup-rank* value as a multiple of 3 will be output.

In summary, just as for slide-by-tuple windows, the WID implementation for these two variations handles disordered input naturally, at the cost of maintaining partial

aggregates for two bins for each tuple. In particular, for the second variation, although its *wids*() function definition uses *rank*() over the *RATTR* attribute, which potentially requires global information over the entire stream, using punctuations removes this "sort" requirement in the implementation. Comparing the WID implementation to the buffering implementation, the major benefit of WID is lower latency for out-of-order input, because WID does not require an ordered stream. We expect that the CPU usage of WID and the buffering implementation are comparable, as they require simlar amount of processing for each tuple. For example, for the second variation, the buffering implementation may need to order the input stream and the WID implementation needs to get the rank for each tuple, both requiring similar processing per tuple.

6.1.5. Performance Study of WID

We tested the effectiveness and efficiency of the WID implementation by conducting three sets of experiments: 1) The first set of experiments compares the execution time performance for sliding windows using the WID implementation and the buffering implementation—the standard implementation for window aggregation, which materializes each window extent and computes the aggregate over it; 2) the second set of experiments compares the latency and accuracy of WID versus the buffering implementation with slack for evaluating queries over streams with *band disorder*; 3) the third set of experiments compares the latency and accuracy WID versus the

buffering implementation with slack for evaluating queries over streams with *block-sorted disorder*. We will introduce band disorder and block-sorted disorder next.

Our experiments were conducted on an Intel® Pentium® 4 2.40 MHz machine, running Linux 7.3, with 512MB main memory. The data size for the experiments was approximately 35 MB.

6.1.5.1. Experimental Data Generation

We implemented a data generator to generate tuples with increasing timestamps loosely based on the XMark data generator [79], which generates online auction data in XML. The first experiment uses the data in generated order. The second and third sets of experiments use data sets with band disorder and block-sorted disorder, respectively.

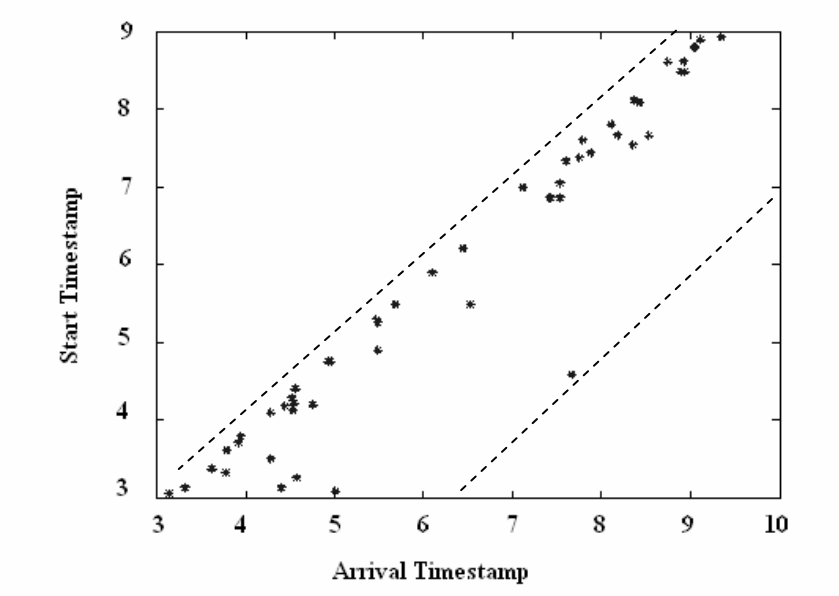


Figure 6-8 Band Disorder—the timestamp of the 8th packet in a NetFlow vs. the start timestamp of the NetFlow

Band disorder and block-sorted disorder are two types of disorder pattern that we observed from network flow data from the the Abilene Observatory, a consortium that uses a high-performance (Internet2) network to study advanced Internet applications [1]. In networking terminology, a network-flow, *NetFlow*, is a connection between a source IP address and port, and a destination IP address and port. A flow comprises one or more packets, which each have a timestamp and size (among other information). Each NetFlow has a start and end time, which are the mininum and maximum timestamps of packets in the NetFlow.

Band Disorder: Figure 6-8 shows the timestamp of the 8th packet in a NetFlow versus the start timestamp of the NetFlow. The relationship between 8th-packet arrival time and flow-start time is near linear, but network delays and packet retransmission result in a "band" of disorder—the dotted lines in the figure shows the band. We call the disorder pattern shown in Figure 6-8 *band disorder*. Many stream systems that handle disorder assume band disorder and handle it with the slack mechanism.

Block-sorted Disorder: Figure 6-9 shows a scatter plot of the stream of all *NetFlow records* emitted by a router in the Abilene Network [1], which exhibit another disorder pattern that we call block-sorted disorder. A NetFlow record is associated with a NetFlow and can be seen as tuple that summarizes the Netflow. The x-axis is the position of the packet in the stream, and the y-axis is NetFlow start time. The graph shows an ascending set of disjoint blocks, with data points scattered apparently at random in each block. The reason for the surprising shape of this graph is that each minute the router outputs all its NetFlow records. At this point, it purges its cache of

NetFlow records and starts over. Thus a block represents the records emitted during a cache purge; the order within a block may be related to the structure of the router hash table. Note that a NetFlow that spans a block boundary is represented as two separate NetFlow records, one in each block. A fixed bound on disorder is not a good match to the disorder pattern shown in Figure 6-9, and thus the slack mechanism will not match it well. Setting the bound to less than a minute will drop many tuples; setting the bound to a minute will accommodate the disorder but unduly delay result output. For example, if the window boundaries match the block boundaries, the disorder here can be well absorbed within individual window extents and thus results need not be delayed at all. What makes more sense is for the router to output a message—a punctuation perhaps—to indicate it has completed a cache purge.

To simulate a band-disorder distribution, we first took ten data sequences (each of

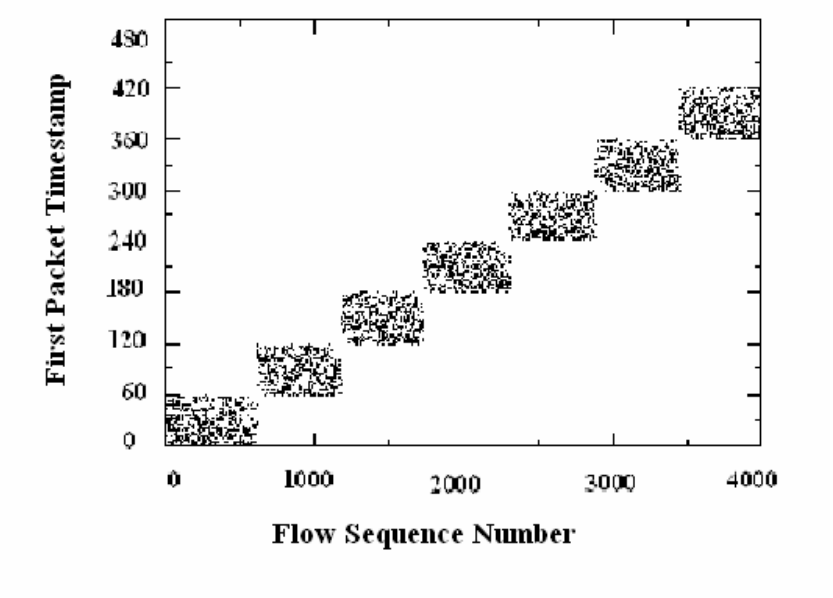


Figure 6-9 Block-sorted Disorder—the arrival position of a NetFlow vs. it start time

them with band disorder) resulting from applying a network-analysis tool [48] over TCP header traces. Each data item in the sequences has a timestamp attribute, which is used for the windowing attribute. To get a long data sequence, we concatenated randomly chosen copies of the ten data sequences. To simulate punctuations from the data source, we pre-processed the disordered data and inserted punctuations into the data. To simulate the block-sorted-disorder distribution, we divided the tuples into segments of equal length on the timestamp attribute, and then randomized the positions of tuples in each segment. We also add punctuation after each "block".

6.1.5.2. Experimental Results

We present the results of the three different experiments in the NiagaraST system. The experiments used variations of Q6-1, and varied the parameters according to Table 6.1. In Table 6.1, Slack Approach includes two flavors of the slack mechanism that we will introduce in the second and the third set of experiments.

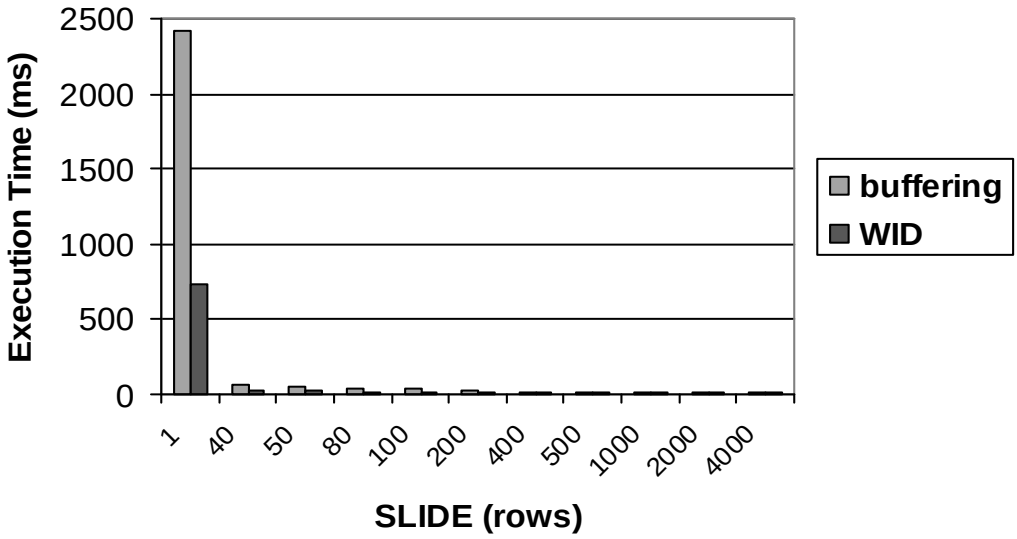
Table 6-1 Experimental Parameters

Exp.	Aggegate Function	Disorder	Slack Size	Slack Approach	RANGE	SLIDE
1	Max	none	0		4000 rows	varies
2	Average	band	varies	consistent generous	64 s	6.4 s
3	Count	block-sorted	varies	consistent	600 s	60 s

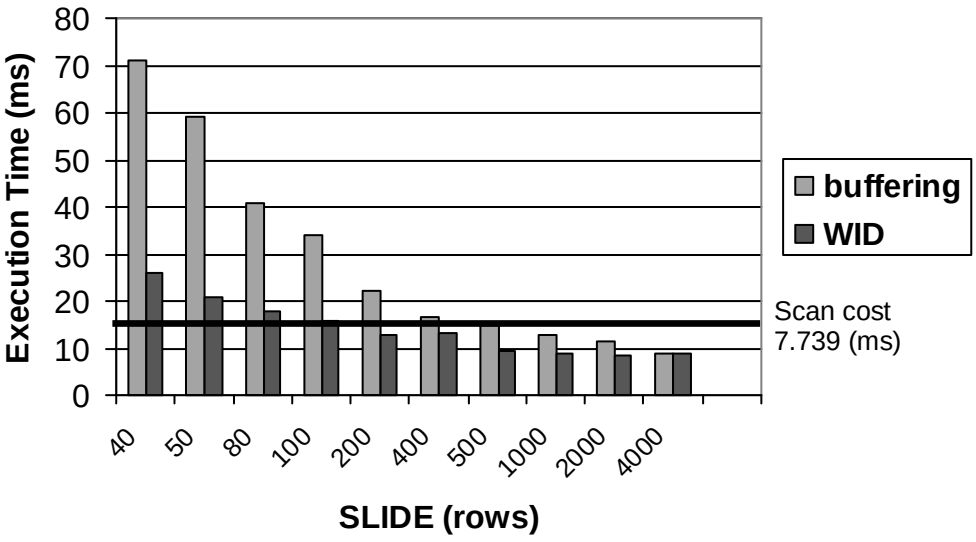
Execution Time Comparison of WID versus Buffering: For Experiment 1, we used the ordered data set and measured the execution time cost of using the WID implementation and the buffering implementation. The measured time is in

milliseconds. For the window specification we used WATTR = row-num, RANGE = 4000 rows, and SLIDE between 1 and 4000 rows. Thus, the number of window extents to which a tuple belongs varies between 4000 and 1. In the experiment, each query is executed for 8 runs and the median of the excution time of the 8 runs is reported here. For each experiment, the system scanned prepared data-files to simulate streams and thus queries in the experiments were executed at the full CPU speed. Therefore, the execution time comparison also directly correlates to CPU-usage and latency performance comparisons.

Experiment 1 (Figure 6-10(a) and (b)) shows that the WID approach in general has lower execution times than the buffering approach; the comparison favors the WID approach as the ratio of RANGE to SLIDE increases. Figure 6-10(b) is a zoomed-in version of Figure 6-10(a), and includes a horizontal line that shows the execution cost of scanning the input stream, which is the measured time of scanning the whole data set.



(a): Execution me: WID versus Buffering – overview



(b): Execution time: WID versus Buffering – zoom-in

Figure 6-10 Execution time comparison using tuple-based sliding-window max, RANGE 4000 rows, SLIDE between 1 and 4000 rows

Latency-Accuracy Tradeoffs for Band Disorder: Recall that the buffering implementation uses slack to handle disorder. For Experiment 2, we used the band-disorder data set and measured the latency-accuracy tradeoff between using punctuation and two types of slack: *consistent* and *generous*. Consistent slack and generous slack are our names for two versions of slack found in the literature [4, 5]. Consistent slack requires that if a late tuple must be dropped from one window extent, it must be dropped from all extents in which it participates, regardless of whether it is late for those other extents or not. Generous slack makes no such restriction. We use mean error percentage as the accuracy metric for this experiment. The aggregate function used in this experiment is average, and mean error percentage is computed as the absolute difference of the true average and the average returned by the query, as a percentage of the true average, over each window extent; then the average of these percentages over all extents is computed. Latency is measured by the wall-clock time between the arrival of a punctuation and the output of the result that the punctuation covers, and we report the average latency over all results of the query. Here, wall-clock time and logical query time are not comparable, because queries are evaluated over streams that are emulated by scanning data files and NiagaraST executes them at maximum speed. The maximum disorder in the data set is 3.2 seconds. For consistent and generous slack, we vary the amount of slack from 0.32 seconds through 3.2 seconds and we use RANGE = 64 seconds, and SLIDE = 6.4 seconds.

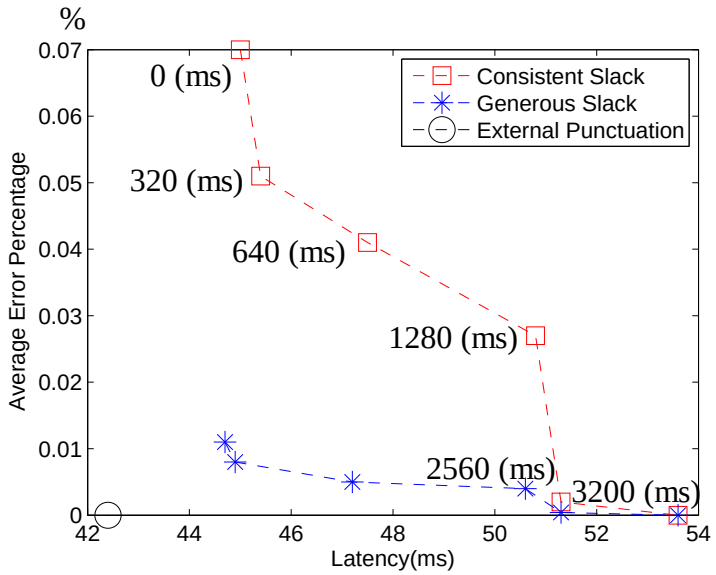


Figure 6-11 Latency-Accuracy (mean error percentage) tradeoff for band disorder: WID vs. Buffering with slack (0ms, 320ms, 640ms, 1280ms, 2560ms, 3200ms along the x-axis) for a window aggregate query (RANGE 64 seconds, SLIDE 6.4 seconds; maximum input disorder is 3.2 seconds

Experiment 2 (Figure 6-11) shows that as slack increases, error decreases and latency increases, as expected. It also shows that external punctuation has better latency and accuracy than either slack mechanism. In addition, generous slack has significantly better accuracy at comparable latency when compared to consistent slack.

Latency-Accuracy Tradeoffs for Block-Sorted-Disorder: Experiment 3 is similar to Experiment 2, except that we used block-sorted disorder with block duration 490 seconds, which means the maximum disorder is up to 490 seconds. We varied the amount of slack from 0 to 600 seconds and used RANGE = 600 seconds and SLIDE = 60 seconds. The aggregate function used in this experiment is Count and we use the

percentage of wrong answers as the accuracy metric for this experiment. The percentage of wrong answers is computed as the number of wrong results over the total number of results that the query produces. In contrast to Experiment 2, where error decreases and accuracy increases as slack increases, for block-sorted disorder there is no linear relationship between slack and latency. For the block-sorted-disorder data set there is one slack value that has the best latency, at the optimal accuracy, as shown in Figure 6-12, which is determined by the relationship between block size and window size. In our experiment, the optimal slack is 491 seconds. When slack is less than optimal, latency is essentially independent of slack. As slack increases above the optimal, latency jumps dramatically. In this case, it would be difficult to use slack to trade off the latency and accuracy of the query as one might hope to do. This experiment also shows that punctuation has better latency and accuracy for block-sorted disorder than any of the slack values used.

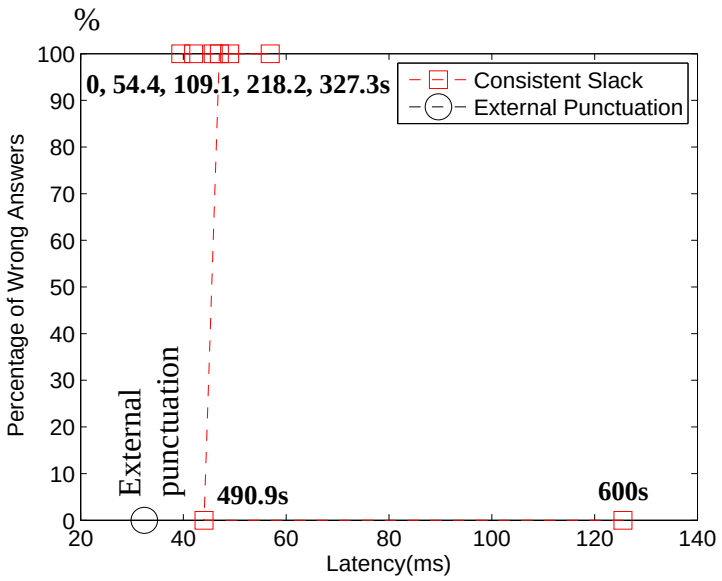


Figure 6-12 Latency-Accuracy (percentage of wrong answer) tradeoff for block-sorted disorder: WID vs. Buffering with slack (0s, 54.4s, 109.1s, 218.2s, 327.3s, 434.2s, 490.9s, and 600s along the x-axis) for a window aggregate query (RANGE 600 seconds, SLIDE 60 seconds); maximum input disorder is 490 seconds

Obviously, the memory usage of the WID implementation usually compares favorably to the buffering implementation. We show a comparision of the memory usage of order-insensitive implementations of window aggregation versus the buffering implementation in section 6.3, when we present our adaptive implementation of window aggregation.

6.2. The Paned-WID Optimization

The computational cost of query evaluation affects CPU usage and thus the throughput of a stream system. In the following, we present an optimization technique for

evaluating sliding-window aggregate queries. This optimization reduces the computation cost by sub-aggregating the input stream and by sharing sub-aggregates among multiple window aggregate computations. The stream is sub-aggregated by non-overlapping sub-sub-streams, which we call *panes*; aggregation over the pane-aggregates is used to compute window-aggregates. The paned optimization can be applied to the buffering implementation, as well.

In the WID implementation of sliding-window aggregation, each tuple belongs to multiple window extents and thus multiple window aggregates are updated with the tuple. For example, without using panes, to evaluate the following query Q6-2, four window aggregates are updated with each tuple, as each tuple contributes to four window extents. As the ratio of RANGE over SLIDE increases, the number of window aggregates updated with each tuple increases. Updating window aggregates can be expensive, especially when the the hash table is large.

Q6-2: "Find the maximum packet size for the past 4 minutes and update the result every 1 minute."

```
SELECT max(length) [RANGE 4 minutes, SLIDE 1 minute, WA ts]
FROM  Main
```

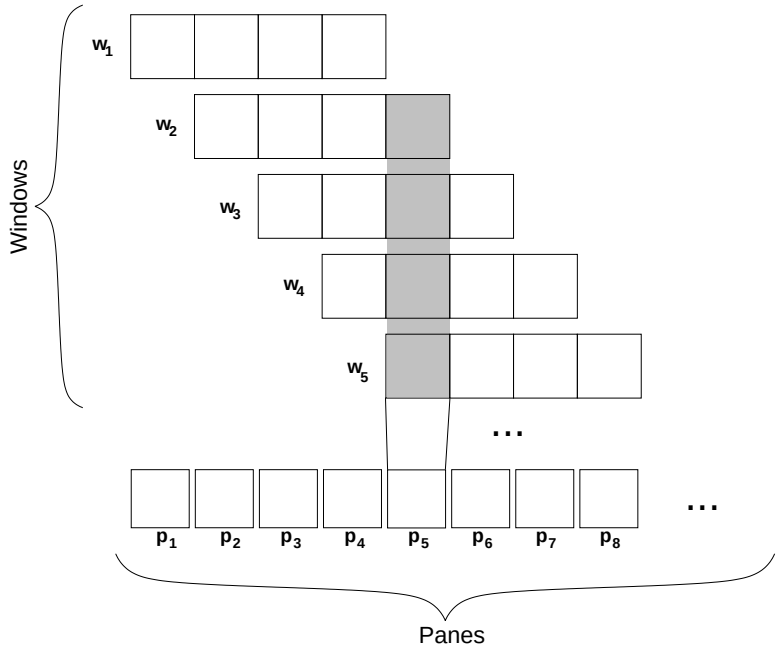


Figure 6-13 Panes for Query 6-2 with RANGE 4 minutes and SLIDE 1 minute; each pane is a 1 minute sub-stream

Figure 6-13 illustrates how panes are used to evaluate Q6-2. The stream is divided into 1-minute non-overlapping panes based on the windowing attribute, *ts*; and each 4-minute window is composed of four consecutive panes. In Figure 6-13, $w_1 - w_5$ are window extents and w_3 is composed of panes $p_3 - p_6$. Each pane contributes to four windows; for example, p_5 contributes to w_2 through w_5. To evaluate Q6-2, we calculate the maximum for each pane; the maximum for each window is computed by finding the maximum of the maxima of the four panes that contribute to the window. For example, the maximum for window w_4 is computed by finding the maximum of the maxima of panes p_4 through p_7.

We note that panes are not always beneficial. For example, for slide-by-tuple windows, panes do not save work because sub-aggregation does not help when each pane contains only one tuple. In general, for a sliding-window aggregate query, the benefit of using panes increases as the number of tuples in each pane increases (i.e., as the average data arrival rate increases).

In the following, we first present the basic structure of the paned-WID evaluation of sliding-window aggregation. Then, we discuss in detail how panes are used for window aggregate queries with different types of aggregate functions.

6.2.1. Evaluating Queries with Panes

To evaluate a sliding-window aggregate query using panes, the query is decomposed into two sub-queries: a pane-level sub-query, PLQ, and a window-level sub-query, WLQ. The PLQ is a tumbling-window aggregate query; it separates the input stream into non-overlapping panes, and produces an aggregate for each pane. The WLQ is a sliding-window query over the result of the PLQ that returns window aggregates.

Figure 6-14 shows the query plan for Q6-2 using panes. Q6-2 is decomposed into a tumbling-window max for the PLQ, consisting of $Bucket^1$ and Max^1 for its execution plan, and a sliding-window max for the WLQ, consisting of $Bucket^2$ and Max^2 for its execution plan. The PLQ produces a pane-maximum for each pane. The aggregates that PLQ outputs have a *pid* attribute, which is the window-id of the aggregate. The WLQ runs over the stream produced by the PLQ, using the *pid* attribute as the windowing attribute, and every minute computes the maximum over the last four

minutes. Typically, PLQ can greatly reduce the data volume of the input stream; for example, for Q6-2, each window (i.e., a 4-minute sub-stream) of the WLQ contains only four tuples, corresponding to four panes.

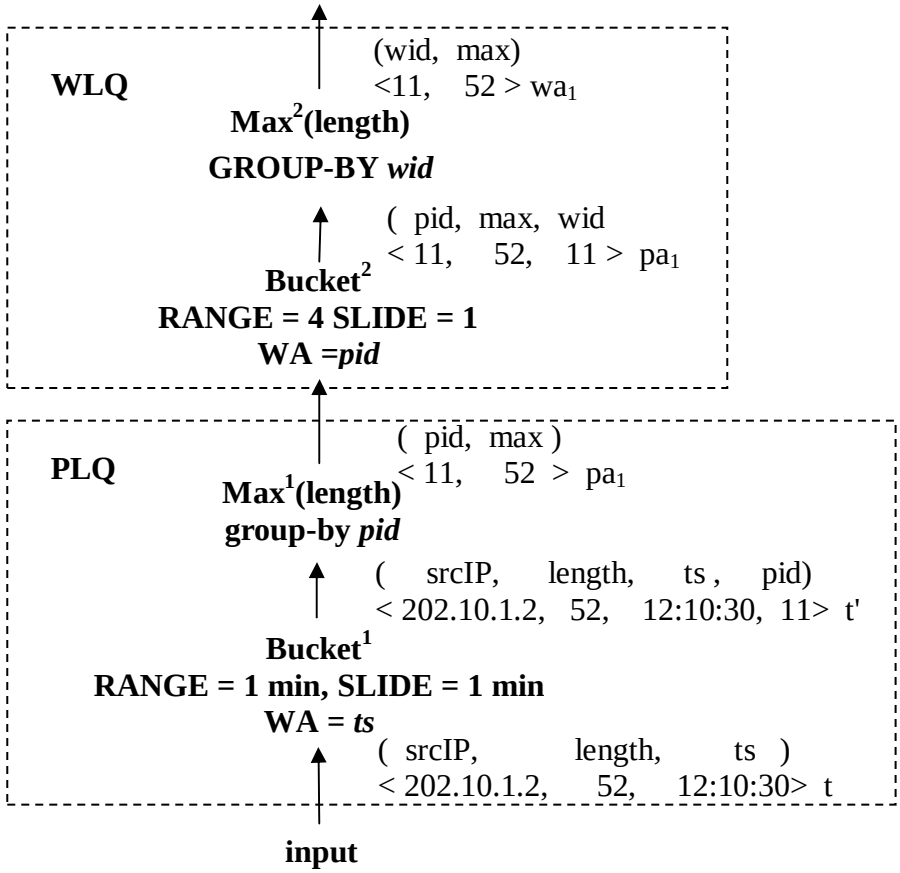


Figure 6-14 Paned-WID for Q6-2 (RANGE 4 minutes, SLIDE 1 minute); PLQ is the pane-level sub-query, and WLQ is the window-level sub-query

In order to use panes, we need to split the original sliding-window aggregate query into PLQ and WLQ, and thus we need to determine the window specifications and the aggregate functions for them. The PLQ and WLQ aggregate functions depend on the aggregate function of the original query. For example, for a sliding-window

maximum, both the PLQ and WLQ use the max aggregate; but for a sliding-window count, the PLQ is a count, and the WLQ is a sum. The window specifications of both sub-queries are also determined by the window specification of the original query. The size of the panes for the PLQ is the largest possible size for sub-aggregation such that the sub-aggregates can be used by the WLQ to compute window aggregates. Therefore, the RANGE, as well as the SLIDE, of the PLQ is the greatest common divisor of the RANGE and SLIDE of the original query: *pane-range* = *pane-slide* = GCD(RANGE, SLIDE). For example, for a window aggregate with RANGE 9 minutes and SLIDE 6 minutes, its pane size (i.e., the RANGE and SLIDE for its PLQ) is 3 minutes. Each window extent of the original query constains 3 panes and consecutive window extents overlap by 1 pane. Thus, for the WLQ of the query, RANGE is 3 and SLIDE is 2, defined on the *pid* attribute of the PLQ results. The WLQ has the same RANGE and SLIDE as the original query, but uses pane-timestamp as the windowing attribute. The number of panes per window is RANGE/*pane-range*. Note that both the PLQ and WLQ are evaluated with WID, and thus using panes does not require any new query operators.

Using panes generally reduces computation cost. Only a single window aggregate is updated for each input tuple in the PLQ. Although multiple window aggregates are updated with each pane-aggregate in the WLQ, the overall computation cost for the query is normally reduced, because the number of panes in a window is usually much fewer than the number of tuples in a window. For example in Query 6-2, each input tuple is processed once to produce a pane-max. Then, each pane-max is used in the

computation of four windows, because each pane-max contributes to four windows. Normally, the number of tuple accesses here is much less than that of accessing each input tuple four times.

6.2.2. Different Types of Aggregates

In the following discussion we introduce two properties of aggregate functions that affect the paned evaluation of sliding-window aggregates.

6.2.2.1. Holistic

Suppose an aggregate function F over a dataset X can be computed from a "sub-aggregate" function L over disjoint datasets X_1, X_2, …, X_n,, where $\bigcup_{0<i\le n} X_i = X$ and a "super-aggregate" function S to compute $F(X)$ from the sub-aggregates, $L(X_i)$, $0 < i \le n$.

$$F(X) = S(\{L(X) \mid 0 < i \le n)\}$$

As defined by Gray et al. [23], an aggregate function F is *holistic* if for all possible sub-aggregate functions, $L()$, there is no constant bound on the size of storage needed to store the result of $L()$. For example, median, quantile, and mode are holistic.

We call aggregates that are not holistic *bounded* aggregates. The term bounded encompasses the distributive and algebraic terms defined by Gray et al. [23]; but the distinction between distributive and algebraic is unnecessary for us. For example, average is bounded: The function $L()$ records count and sum; the function $S()$ adds the

respective components and then divides to produce the global average. Other common examples of bounded aggregates include count, max, min, sum, variance, and center-of-mass.

6.2.2.2. Differential Aggregate Functions

We define the differential[3] property for aggregate functions. Assume there exist two datasets X and Y such that $Y \supseteq X$. Aggregate F is *differential* if there exist functions L, H and J that satisfy the two conditions: 1) $F(Y - X)$ can be computed from $L(Y)$ and $L(X)$ and 2) $F(Y)$ can be computed from $L(Y - X)$ and $L(X)$ as below:

$$F(Y - X) = H(L(Y),\ \ L(X))$$
$$F(Y) = J(L(Y - X),\ \ L(X))$$

We also require that $|L(X)| < |X|$.

For example, count is differential as shown below.

$$count(X' - X) = count(X') - count(X)$$
$$count(X') = count(X' - X) + count(X)$$

Based on the sub-aggregate function L, we further categorize differential aggregate functions. If the result of L can be stored with constant storage, we say that F is *full-differential*. For example, count, average and variance are full-differential. A full-differential aggregate function must be bounded. If the result of L cannot be stored with constant bound, we say that F is *pseudo-differential*, for example, a heavy-hitter aggregate that finds frequently occurring items is pseudo-differential, because

[3] Differential is similar to what Arasu and Widom term *subtractable* [5].

although an L function exists for heavy-hitter, the result of the L function cannot be stored with constant storage.

Next, we discuss using panes to evaluate bounded and holistic aggregates, respectively. We also discuss the effects that the differential property and the number of groups have on evaluating sliding-window aggregate queries.

6.2.3. Paned-WID for Queries Using Bounded Aggregate Functions

For a differential aggregate function, we can exploit the differential property to further reduce its evaluation cost by computing the aggregate for the current window based on the aggregate of the previous window. For example in Q6-2, to compute the count over w_3 as shown in Figure 6-13, we can use $count(w_3) = count(w_2) - count(p_2) + count(p_6)$. To leverage the differential property, the aggregate operator (in the WLQ) needs to handle tuple deletion, as well as tuple insertion.

The GROUP-BY construct introduces another factor, the number of groups, into the space requirement and computation cost. Intuitively, the more groups, the more space and the more computation are needed to evaluate the query. The following query Q6-3 is a sliding-window aggregate query with GROUP-BY.

Q6-3: "Count the number of packets from each source IP for the past 4 minutes and update the result every minute."

```
SELECT count(*) [RANGE 4 minutes, SLIDE 1 minute, WA ts]
FROM packets
GROUP BY srcIP
```

Using panes to evaluate Q6-3, every group in each pane is aggregated into a <*srcIP*, *pane-count*, *pid*> tuple by the PLQ. Assuming *G* groups per pane, for the WLQ, a window contains 4**G* tuples, as there are four panes per window. The number of groups per pane, *G*, is important because for each group the PLQ constructs an output tuple and the WLQ processes an input tuple. In the extreme case where every group contains a single tuple, the PLQ cannot reduce the number of input tuples for the WLQ and panes provide no benefit. In fact, for a bounded aggregate query with a GROUP-BY, the size of the required space is bounded only if the number of groups is bounded.

Taking both the number of groups and the differential property of the aggregate function into account, we express the computational cost per window-aggregate of using panes for sliding-window queries with non-differential and differential aggregate functions, $Time_{P\text{-ND}}$, and $Time_{P\text{-D}}$. In the following discussion, we use count and maximum as the representative for differential and non-differential aggregate functions, respectively.

$$Time_{P\text{-ND}} = a*T/P + b*G + c*P*G \quad \text{(Eq. 6.1)}$$

$$Time_{P\text{-D}} = a*T/P + b*G + 2*c*G*SLIDE/\text{GCD}(RANGE, SLIDE) \quad \text{(Eq. 6.2)}$$

In the two formulas above, *a* is the PLQ's cost to process an input tuple, *b* is the PLQ's cost to generate an output tuple, and *c* is the WLQ's cost to process a tuple (to add a tuple to a window (e.g., to update the maintained aggregate with the tuple) or to remove a tuple from a window (e.g., to subtract the tuple from the maintained aggregate for queries using differential aggregate functions such as count), *T* is the

number of tuples per window, *P* is the number of panes per window, and *G* is the number of groups per pane. In Eq. 6.2, *SLIDE*/GCD(*RANGE*, *SLIDE*) is the number of panes per slide. For example, when range is 9 minutes and slide is 6 minutes then the pane size is 3 minutes, so the number of panes per slide is 2. Thus, 2**c***G***SLIDE*/GCD(*RANGE*, *SLIDE*) is the cost to compute the aggregates for all groups in the current window based on the aggregates in the previous window, that is, the cost to expire old panes and the cost to add new panes.

The cost per window of evaluating sliding-window queries with non-differential and differential aggregation functions without using panes, $Time_{W\text{-}ND}$ and $Time_{W\text{-}D}$, are as follows, where a' is the cost to process each tuple (to insert a tuple to or to remove a tuple from a window).

$$Time_{W\text{-}ND} = a'*T \quad \text{(Eq. 6.3)}$$

$$Time_{W\text{-}D} = 2*a'*SLIDE*(T/RANGE) \quad \text{(Eq. 6.4)}$$

Without using panes, the WID implementation for a sliding-window query with a non-differential aggregate function such as maximum needs to use every tuple in the window extent to compute its aggregate, just as Eq. 6.3 indicates. The WID implementation cannot directly leverage the differential property, because leveraging the differential property requires processing window extents sequentially. As we assume linear punctuation, pane results are produced in order, and thus WLQ receives an ordered stream. Eq. 6.4 shows the computational cost of using the buffering implementation to evaluate a sliding-window query with a differential aggregate function, such as count. The buffering implementation can compute the count for the

current window extent based on the count of the previous window extent by adding one to the previous window-count for each new tuple for the current window and subtracting one for each expired tuple.

Comparing Eqs. 6.1 to 6.3, and 6.2 to 6.4, we see that there are some situations in which using panes might not provide performance gains: 1) When the number of groups per pane increases above a certain threshold; and 2) when the number of panes per window is too small, for example, one pane per window.

6.2.4. Panes for Queries Using Holistic Aggregate Functions

Similarly, for holistic aggregates, the pre-processing of panes can also be shared by multiple windows to reduce computation cost. We use heavy hitters as a holistic-aggregate example, and use a algorithm that is similar to that used by Gigascope to evaluate heavy hitters.

In Giagasope, to evaluate heavy hitter queries such as “find the IP sources that most frequently generate packets”, multiple alternatives are available for sub-aggregate and super-aggregate pairs [13]. One option is that the sub-aggregate uses a hash table to record the packet-count for each IP source, and then the super-aggregate uses the hash table entries to update its data structure, called a *sketch*, for estimating heavy hitters. Although Gigascope only evaluates tumbling windows, we can use a similar method to evaluate sliding-window heavy hitter queries, such as Q6-4.

Q6-4: “Over the past 10 minutes, find the srcIPs from which the number of packets received is greater than or equal to 5% of the total number of packets received; update the result every minute.”

To evaluate Q6-4, the PLQ maintains a hash table with (srcIP, count) hash entries. At the end of each pane, the non-empty hash table entries are output. The WLQ buffers and uses each hash table entry to update the sketches for multiple windows. Using panes, the PLQ compresses all the packets from a source IP to a single hash entry and reduces required buffer spaces, similar to the sub-aggregation in Gigascope. In addition, each hash table entry is used by multiple windows, and thus reduces the overall computation cost. Similar strategies can be applied to evaluate other sliding-window holistic aggregates using panes.

We note that in order to use panes, differential holistic aggregate functions need necessarily be pseudo-differential. Consider heavy hitters: The counts recorded by hash table entries can be summed or subtracted. Thus, the sketch of the current window can be constructed based the sketch of the previous window; but there is no bound on the number of hash entries for each pane, as the number depends on how many groups are represented in the pane.

6.2.5. Performance Study of Paned-WID

We experimentally compared the execution of sliding-window aggregate queries with and without panes. Our experiments were conducted on an Intel® Pentium® 4 2.40 MHz machine, running Linux 7.3, with 512MB main memory. Our data generator is loosely based on the XMark data generator [79], and the data size for the experiments was approximately 15.2 MB. We calculated execution time by measuring the query execution time and then subtracting the cost of scanning the input stream, to focus on just the aggregation cost.

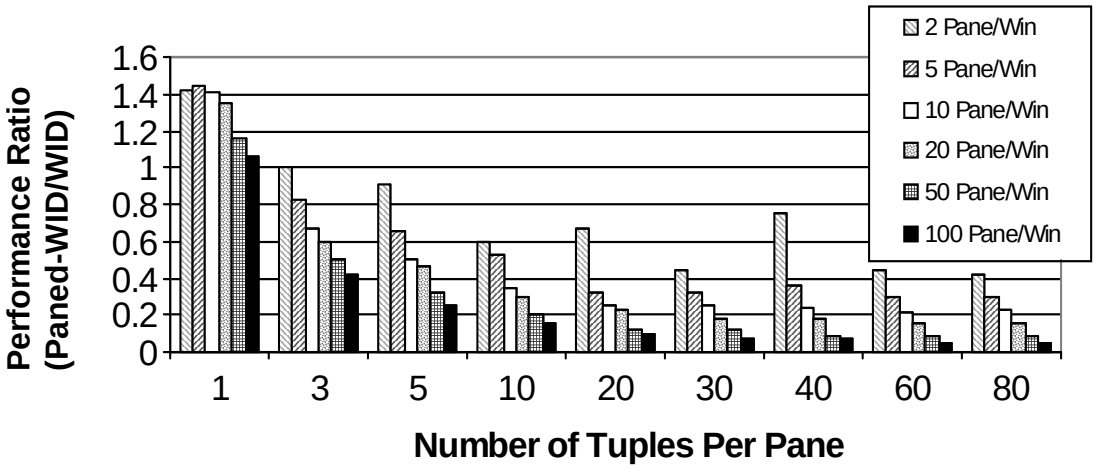


Figure 6-15 Execution-time ratio of the Paned-WID vs. the WID for a sliding-window maximum query (varying the number of tuples per pane and the number of panes per window)

In our experiments, we varied the RANGE and the SLIDE parameters of a sliding-window max query, Q6-2, effectively varying the number of tuples per pane, and the number of panes per window (i.e., Pane/Win, as shown by the different columns of each group in Figure 6-15). Figure 6-15 shows the ratio of the execution time using

panes over the execution time of the WID implementation without panes. For example, we see that at 20 tuples per pane and 5 panes per window, the paned option takes about 30% of the time of the non-paned option. We conclude from Figure 6-15 that using panes has better execution-time performance than the original approach in many cases.

We expect that the memory usage of Paned-WID will be similar to that of WID, as both of them maintain partial aggregates. The PLQ in Paned-WID only maintains one aggregate for each group. The number of aggregates that WLQ maintains for each group is the same as that of WID if SLIDE evenly divides RANGE; if not, WLQ needs to maintain more partial aggregates than WID for each group.

6.3. The AdaptWID Implementation

Memory performance is important for processing high-volume data streams. Compared to the buffering implementation, the WID implementation is often more memory efficient because maintaining partial aggregates normally requires significantly less memory than buffering tuples. However, this memory-usage difference is directly dependent on physical stream properties. For example, consider a network-packet stream with the simplified schema of <*srcIP*, *destIP*, *ts*> and the following window aggregate query, Q6-5, with the timestamp attribute of the packet stream, *ts*, as the windowing attribute (WA).

Q6-5: "Count the number of packets from each source IP for the past 5 minutes; update the result every 1 minute"

```
SELECT srcIP, count(*)
                [RANGE 5 minutes, SLIDE 1 minute, WA ts]
FROM M1
GROUP BY srcIP
```

The buffering implementation maintains a buffer of 5 minutes of tuples (i.e., a window extent) at all times. At the end of each minute, it computes and outputs the number of packets from each srcIP over the buffered tuples, and purges the one minute of expired tuples from the buffer. The WID implementation maintains the number of packets from each source IP for each active window extent. For Q6-5, if the input stream is ordered, there are five active windows at a time; the WID implementation incrementally computes the number of packets from each source IP for the five active window extents. At the end of a window extent, the WID implementation outputs the aggregates for that window extent and then purges those aggregates. Although the WID implementation is normally very memory efficient, the buffering implementation may use less memory for a given srcIP when the input stream is very sparse. For example, if some group of Q6-5 contains only one tuple every five minutes, the WID implementation needs to maintain 5 partial aggregates, whereas the buffering implementation would only buffer one input tuple for the group. Thus, the WID implementation is memory efficient when groups are dense; that is, each group has many tuples per window. If there are too many sparse groups, the WID method may have excessive memory overhead compared to the buffering implementation. For ease of presentation, we term the buffering technique and the WID implementation as *lazy* and *eager* aggregation, respectively.

Window-aggregate evaluation may benefit from a hybrid of eager and lazy aggregation. Massive data streams can often exhibit data skew, with a tail of many sparse groups in addition to a small number of dense groups. Data-distribution skew (e.g., a power-law distribution in group density), which often occurs with high data volumes, may lead to a large proportion of sparse groups. For example, distribution of network packets at a router is highly skewed, with a large number of packets coming from a small set of IP addresses, but a few packets coming from each of many other IP addresses. For some critical scenarios, such as denial-of-service attacks, the percentage of small groups increases dramatically. Processing sparse groups with lazy aggregation and dense groups with eager aggregation may lead to better memory performance than either lazy or eager aggregation alone. Further, stream systems generally cannot statically differentiate sparse groups from dense groups, and the character of a group can change over time. For example, the number of bids for an online auction item may change dramatically over time: An auction might not receive many bids until its expiration time is approaching. Thus, the system needs to determine dynamically at execution time which aggregation method to use and provide an adaptive mechanism to switch between the aggregation methods.

We examine stream properties that affect memory efficiency of window aggregation and propose an adaptive implementation, *AdaptWID*, that combines the best aspects of the buffering implementation and the WID implementation to improve memory efficiency for input streams with skewed data distributions, even if the distributions

vary over time. AdaptWID adapts the aggregation method on a group-by-group basis to cope with time-varying data skew.

In the following, we first discuss input stream properties that affect memory efficiency of the lazy and eager implementations and use them to model their memory usage. AdaptWID uses this memory-usage model to select among evaluation algorithms at run time. Then, we present the AdaptWID implementation. Our experimental study verifies that the adaptive algorithm improves memory usage, while maintaining execution cost and latency comparable to existing non-adaptive implementations.

6.3.1. Stream Properties and Memory-Cost Estimation

To allow the aggregate operator to choose between eager and lazy aggregation, we need to estimate the memory costs of both implementations for each group. In the following, we discuss stream properties that may affect memory usage and then present memory-cost estimates for lazy and eager aggregation based on these properties. We assume input streams may contain data from multiple sources, may contain out-of-order tuples and are punctuated. The metrics that we propose for measuring stream properties are defined relative to the window parameters (RANGE, SLIDE, WA). Note that an ordered stream is a special case of our stream model and the discussion here applies.

Stream Volume: Stream volume describes the amount of data in the stream. We define stream volume, *vol*, as the amount of data per unit of the windowing attribute. For example, if the unit of the windowing attribute is seconds, then *vol* is the number

of tuples with windowing attribute values within a given second. Notice that stream volume is determined only by data in the stream; it is independent of the stream-arrival pattern, and is different from the real-time stream-arrival rate. Stream-arrival patterns and rates are affected by data transmission, and may fluctuate even when stream volume is stable.

Arrival Order: Out-of-order tuples delay the completion of window extents in which they participate and thus increase the number of window extents open at a time. We measure a stream's arrival order by window-extent duration, *wed*, which is defined on the windowing attribute and is the length of the period that a window extent is active. Let the *high-watermark* of a data source be the largest value of the windowing attribute seen so far in the stream; and let the *low-watermark* be the smallest value of the windowing attribute that might still appear in the stream. (Note that low-watermark indicates the progress of the stream.) We define the *wed* of a window extent as the difference between the low-watermark at the start of the extent and high-watermark at the completion of the extent. An extent starts on the arrival of the first tuple belonging to the extent and completes on the arrival of the punctuation covering (closing) the extent. Intuitively, a longer *wed* indicates more window extents open simultaneously.

Arrival-Time Skew: When the input stream consists of data from multiple data sources, skew in the arrival times of different sources can cause disorder in the combined stream. *Arrival-time skew* describes the time skew among data sources. We measure the synchronization of two data sources at each instant by *offset*, which is the

difference between the high-watermarks of the sources. As we will see later, knowing the offset helps to better estimate the state requirements of window operators.

6.3.2. Memory-Cost Functions

In general, for aggregation, the total amount of state that must be maintained is determined by the number of open window extents and the amount of state maintained for each open extent. An open window extent is one that has started (with its first tuple arrival) but not completed (with covering punctuation arrival). AdaptWID needs to process each group individually, and thus requires memory-cost estimates for window aggregation for each group.

The memory cost of eager aggregation is determined by the number of open window extents and the size of the partial aggregate. Given the duration of a window extent, *wed*, the number of open window extents is *wed*/SLIDE. The memory cost of eager aggregation is given by Eq. 6.5 below, where *aggr* is the size of a partial aggregate.

$$MC_{eager} = aggr * (wed / SLIDE) \qquad \text{(Eq. 6.5)}$$

The memory cost of lazy aggregation is determined by the number of buffered tuples. Consider the sub-stream for one group and assume that the stream volume, *vol*, is relatively constant over the duration of a window extent. Eq. 6.6 below estimates the memory cost for lazy aggregation, where *tup* is the size of an input tuple.

$$MC_{lazy} = tup * vol * wed \qquad \text{(Eq. 6.6)}$$

However, if the input to the aggregate is the union of multiple sources, arrival-time skew of these sub-streams affects the estimation of MC_{lazy}. Figure 6-16 shows the

synchronization of three data sources, A, B, and C. Assume the sub-streams from each data source arrive in order. The points $open_i$ and end_i mark the active periods of a window extent for i = A, B, C. The close point marks the arrival of the punctuation covering this window extent. Thus, the interval between $open_i$ and the close point corresponds to the *wed* for each source. In this example, tuples in source A arrive earlier than B and C, and $offset_{B\rightarrow A}$ and $offset_{C\rightarrow A}$ respectively, indicate the skew of B and C relative to A, respectively. The duration of the whole window extent is marked by *duration*, which is the same as the *wed* for the earliest source, A.

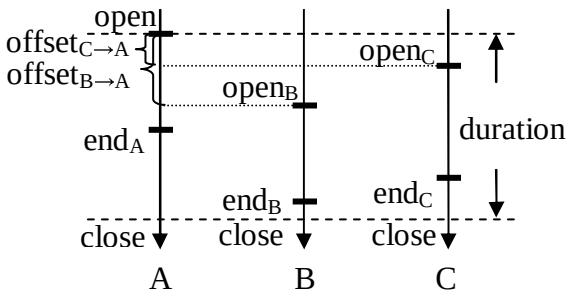


Figure 6-16 A window extent for unsynchronized data sources A, B, and C

The input tuples that lazy aggregation needs to buffer include all the tuples within the *wed* period of each data source. Assume that vol_i is the volume of stream i. The number of tuples that lazy aggregation buffers is

$$(vol_A * wed_A + vol_B * wed_B + vol_C * wed_C)$$

Here, wed_A equals the duration of the full window extent wed, wed_B is ($wed - offset_{B\rightarrow A}$), and wed_C is ($wed - offset_{C\rightarrow A}$). Letting *vol* equal ($vol_A + vol_B + vol_C$), the number of tuples buffered by lazy aggregation is

$$vol * wed - (vol_B * offset_{B\rightarrow A}) - (vol_C * offset_{C\rightarrow A})$$

The general formula for n data sources, S_1, S_2, …, S_n is as follows, assuming that S_1 is the earliest-arriving data source.

$$numTuples = \left(vol * wed - \sum_{K=S_1 \ldots S_n} vol_K * offset_{K \to S_1} \right) \qquad \text{(Eq. 6.7)}$$

The memory cost of lazy aggregation is then given by Eq. 6.8.

$$MC_{lazy} = tup \left(vol * wed - \sum_{K=S_1 \ldots S_n} vol_K * offset_{K \to S_1} \right) \qquad \text{(Eq. 6.8)}$$

Given the memory costs in Eq. 6.5 and Eq. 6.8 for eager aggregation and lazy aggregation, respectively, we derive a threshold condition to indicate when eager is preferred over lazy aggregation as shown in Eq. 6.9; *winVol* equals *vol * wed*, which we call *window volume*.

$$winVol > \frac{aggr * wed}{tup * SLIDE} + \sum_{K=S_1 \ldots S_n} vol_K * offset_{K \to S_1} \qquad \text{(Cond. 6.9)}$$

The aggregate operator in the AdaptWID implementation actively monitors the stream properties used in Cond. 6.9 for each group, and triggers the switching between eager and lazy aggregation for the group based on the threshold condition. The memory-cost models for eager and lazy aggregation make it possible to estimate which will use less memory for given input-stream properties.

6.3.3. The Runtime Switching Mechanism

Efficient runtime switching between lazy and eager aggregation is essential for AdaptWID. The AdaptWID implementation actively monitors the stream properties of

each group to estimate memory costs using the memory-usage model, and based on the estimates, it will switch between eager and lazy aggregation for a group. We first discuss runtime switching in AdaptWID in this section, then present the detailed implementation in the next section.

Efficient runtime switching requires a low switching cost and a short transition period. In the following we discuss switching between eager and lazy aggregation in either direction for a single group.

Lazy to eager: Switching from lazy to eager aggregation is straightforward. The aggregate operator uses the buffered tuples to construct partial aggregates, and then discards those tuples. Tuples arriving during the transition are processed immediately without buffering.

Eager to lazy: Switching from eager to lazy aggregation is more challenging, because we cannot reconstruct tuples from partial aggregates, nor discard those aggregates immediately. Therefore, during the transition in this direction, we must maintain both the partial aggregates computed so far and the tuple buffer for new input, until all existing partial window aggregates are output. The number of partial aggregates for each group is at least the number of window extents in which each tuple participates, which is determined by the window specification. For example, in Q6-5, at least five partial aggregates are maintained for each group. Out-of-order tuples may increase the number of partial aggregates that the query needs to maintain. However, for a tumbling-window query, there will often be only one partial aggregate for each group, and thus that transition cost is lower than that for sliding-window queries.

Leveraging panes: The overhead for switching from eager to lazy can be improved in both the length of the transition period and the amount of memory usage by using panes. For example, to evaluate Q6-5 using panes, a sliding-window count query with RANGE 10 minutes and SLIDE as 1 minute, the query is split into a sub-aggregation with a 1-minute tumbling-window and a sub-aggregation computes the count for a window extent of the query by summing up the results of the sub-aggregations on panes. The tumbling-window sub-aggregate will often have a much lower switching cost than the original query. Although the super-aggregation is a sliding-window aggregate, the number of tuples in a window extent of super-aggregation is bounded ($\leq$ 10 in this case), and thus lazy aggregation is a good choice for it, and adaptive switching is not needed.

6.3.4. Implementation Details

Like the WID implementation, the AdaptWID implementation also has two parts, a Bucket operator and an Aggregate operator. The bucket operator is the same as in the WID implementation. The aggregate operator processes each group independently using either eager or lazy aggregation, and may switch between them during execution. Switching is governed by the threshold condition defined in Cond. 6.9, which requires monitoring relevant properties of each group, as we expect both data and streaming properties to change over time in many applications.

In the following, we present the AdaptWID implementation. Consider a sliding-window query, Q6-6, as a running example in the presentation. The aggregate operator receives linear punctuation on the *wid* attribute.

Q6-6: "Computes the total size of the packets in the past 10 seconds for each source IP and update the results every second."

```
SELECT srcIP, sum(len)
                [RANGE 10 seconds, SLIDE 1 second, WA ts]
FROM packets
GROUPBY srcIP
```

6.3.4.1. Monitoring Stream Properties for Switching

AdaptWID needs to determine the stream properties involved in the threshold condition for each group. Window volume, *winVol*, is determined by keeping a count of the number of active tuples in a window extent for each group. Window extent duration, *wed*, is initiated to RANGE, and is updated at each punctuation with the *wed* of the most recently closed window extent—the difference between the high-watermark of the completion and the low-watermark of the start of the extent. High-watermark is estimated by the largest *ts* value among the input tuples, and low-watermark is estimated by the largest *ts* value of punctuations. If tuples are produced from multiple data sources, the threshold is also affected by the offset among the data sources, and the stream volume of each data source. In many applications, the offset values among data sources are relatively static and can be pre-determined; otherwise, the offset between any two sources can be estimated by the difference between the data sources' high-watermarks. The stream volume of each data source can be

deduced by the number of tuples in a window extent from each data source divided by the RANGE parameter of the window.

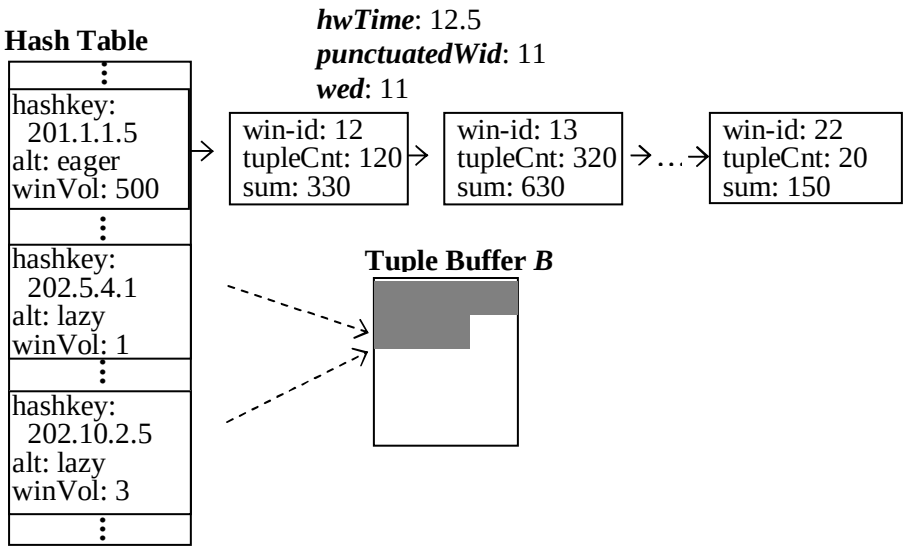


Figure 6-17 The AdaptWID Evaluation of Q6-6 with RANGE 10 seconds and SLIDE 1 second—dense groups are evaluated with eager aggregation and sparse groups are evaluated wth lazy aggregation

6.3.4.2. Implementing the Aggregate Operator

As the Bucket operator of AdaptWID is the same as that of WID, we focus on the Aggregate operator. The state that the Aggregate operator maintains for AdaptWID is more complex than for WID. Figure 6-17 shows the data structure and state that AdaptWID maintains during evaluating Q6-6. To support both eager and lazy aggregation, the aggregate operator maintains a hash table, H, and a tuple buffer, B. Each group has an entry, g, in H. For an eager group, g contains a list of partial window aggregates, one for each active window extent in the group. Notice that the

counter *tupleCnt* keeps track of the number of tuples that will expire when the aggregate is released, instead of the total number of tuples in the window extent. For a lazy group, g indicates this status by the value of the flag *alt* and all the input tuples for g go in to the shared buffer B.

The Aggregate operator also maintains the following state for each data source (Figure 6-17 assumes a single data stream): *punctuatedWid*, the last punctuated *wid* value; *hwTime*, the high-watermark time of the input stream, which is initialized to 0 and updated to max(*t.ts*, *hwTime*) as each tuple *t* arrives; and *wed*, the duration of the last completed window extent, which is initialized to RANGE, and is updated when the window extent completes, with the difference of the current *hwTime* and the stream high-watermark when the window extent starts.

The aggregate operator in AdaptWID processes two kinds of input, tuples and punctuations. In addition, tuple arrival may cause a lazy group to switch to eager, and punctuation arrival may cause an eager group to switch to lazy. We discuss tuple processing, switching, and punctuation processing separately below.

Processing Tuples: When a tuple *t* arrives, the aggregate operator hashes *t* on its grouping values to locate its hash entry g. There are three possibilities.

1. Entry g is null (i.e., no existing group for *t* in H): Create an entry for a new lazy group in H, with *winVol* = 1 and the lazy alternative selected. Buffer *t* in B. Note that initially, every group is lazy.

2. Entry g contains a lazy group: Add *t* to B and increment *winVol*.

3. Entry g contains in an eager group:

3.1. Update all partial aggregates in *t*'s group that match *t*'s *wid* range. If the window-id of a partial aggregate equals the upper bound of *t*'s *wid* range, increment its *tupleCnt*. Increment g.*winVol*.

3.2. Create new partial aggregates in g for any later extents to which *t* belongs in g. Notice that the counter *tupleCnt* keeps track of the number of tuples that will expire when the aggregate is released. Thus, initialize *tupleCnt* of a partial aggregate to 1 if *t* does not belong to any later window extent; otherwise, initialize *tupleCnt* to 0, because if *t* belongs to later window extents, it should not be expired when the current aggregate is released.

Switching: Tuple arrival may switch a lazy group to eager, if *winVol* rises above the threshold. To switch, the Aggregate operator scans B, using tuples in the group to build partial window aggregates, and sets the status indicator, *alt*, of the group to "eager". For tuple *t* with wid range *i* to (*i* + *n*), we update extents from max{*punctuatedWid* + 1, *i*} to (*i* + *n*). Here, *punctuatedWid* records the window-id of the last completed window extent, and thus window *punctuatedWid* + 1 is the first active window extent.

Punctuation arrival may switch a group from eager to lazy, if input tuples expire and winVol decreases below the threshold. When that happens, the Aggregate operator marks the group as lazy and puts subsequent input tuples into B, but still maintains existing partial aggregates for the group until those aggregates are all output. Such a group is called a transitional group. If winVol for a group fluctuates around the threshold, the group could oscillate between eager and lazy. To avoid such thrashing,

we set two threshold values, one for switching from lazy to eager aggregation, and a slightly lower one for switching back.

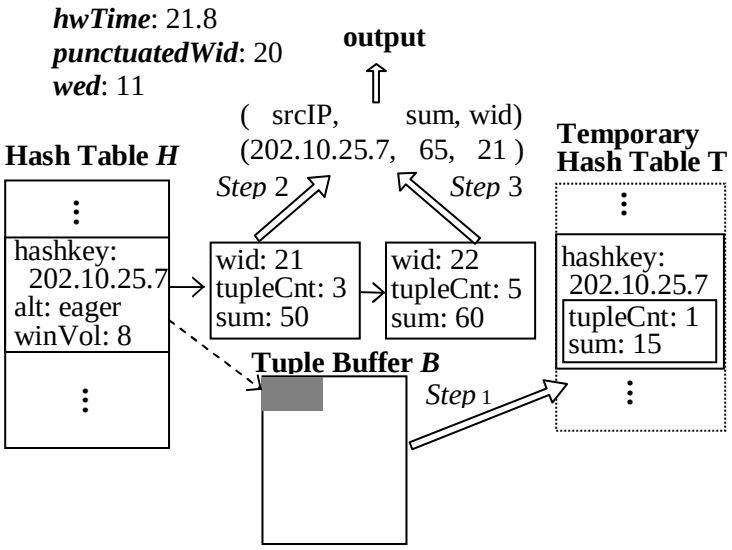


Figure 6-18 Outputting a result for a group in transition—a result is produced with data from both hash table H and the temporary hash table built to compute aggregates from tuples in buffer B

Processing Punctuation: Punctuation arrival will trigger output of window aggregates for completed window extents, and the aggregate operator processes each group according to its status—eager, lazy, or transition—as follows.

1. Eager: Scan H to find all eager groups. For each such group, remove and output partial aggregates covered by the punctuation, then decrease *winVol* of the group by the *tupleCnt* of each such partial aggregate.

2. Transition: Figure 6-18 shows punctuation processing for a group in transition. Scan B, using tuples that match the punctuation to build a temporary hash table T on the grouping attributes. Remove tuple *t* if the punctuation covers the upper range of its

wid, and decrement *winVol* for its group. Scan T and output the aggregate for each group, merging it with the partial aggregate for the same group and *wid* in H, if such exists. (In practice, we reuse H for the temporary hash table T, to avoid the overhead of building another hash table.)

3. Lazy: Process as in Case 2, except there are no existing partial aggregates to merge.

At any time, if *winVol* of a group drops to 0, remove it from H.

Discussion: A potential problem here is that the memory cost in the transition period might be higher than with either eager or lazy alone, and the transition period lasts for almost the duration of a window extent. However, we expect only a fraction of groups to be in transition at the same time. Another possible problem is that all the lazy groups share buffer B. As the the number of tuples in B increases, the latency for lazy to eager transition increases, because it requires scanning B to find tuples belonging to the group switching to eager. To reduce that latency, we could partition the shared buffer B into bins, and partition the hash table H into corresponding sections, and let groups in each section share one bin. Note that because only tuples from sparse groups go to buffer B, the size of buffer B is linear in the number of groups. The AdaptWID implementation can be enhanced with panes, as we have discussed. The tumbling-window sub-aggregation can designate individual panes in a group as eager or lazy. If the collective size of the tuples in the pane is greater than a pane aggregate, the item contains a partial aggregate (eager); otherwise, it contains a list of input tuples (lazy).

6.3.5. Performance Study of AdaptWID

We implemented AdaptWID and compared it to eager and lazy aggregation in NiagaraST. All of our experiments were conducted on an Intel® Pentium® 4 3.40 GHz machine, running Linux (Centos 7.3), Sun® Java VM 1.5, with 1GB main memory. We used two queries in this part of the performance study, Q6-7 and Q6-8. Q6-7 is a tumbling-window sum query with window size one second, and thus represents tumbling-window aggregation over single data source; Q6-8 is a sliding-window count query over the union of three network links and thus represents sliding-window aggregation over multiple data sources. We assume linear punctuation on *ts*. Thus, window extents of all groups are terminated at the same time.

Q6-7: "Compute the total size of the packets from a network traffic link, Main, in the past 10 seconds for each source IP; update the results every second."

```
SELECT srcIP, sum(len)
                [RANGE 1 second, SLIDE 1 second, WA ts]
FROM Main
GROUP-BY srcIP
```

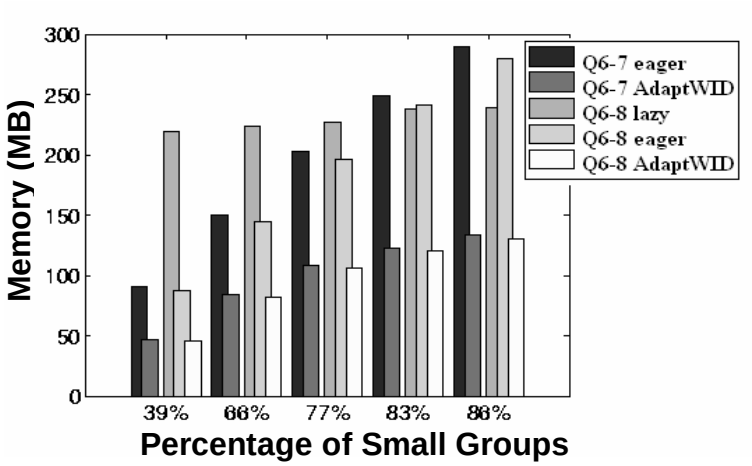
Q6-8: "Count the number of the packets from three network traffic links, Main1, Main2 and Control, in the past 10 seconds for each source IP, and update the results every second."

```
SELECT count(*)
        [RANGE 10 sec, SLIDE 1 sec, WA ts]
FROM (Main2 ∪ (Main1 ∪ Control))
GROUP-BY srcIP
```

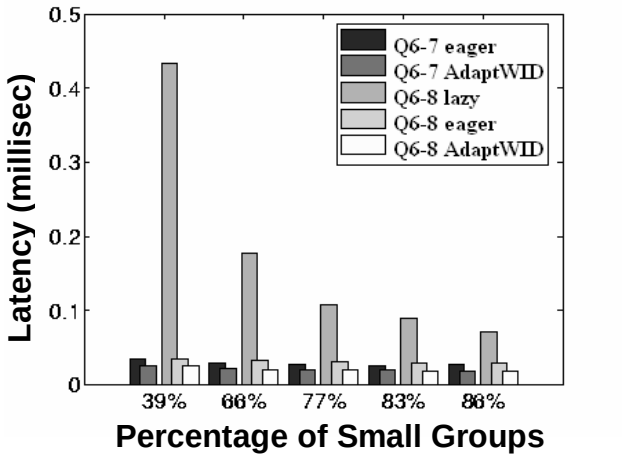
Table 6-2: Five Data Sets (DS1 – DS5) with Skewed Data Distribution—Each contains a different percentage of small, medium, and large groups. (The small groups of DS1 – DS5 contain 1, 3, 5, 7, 9 percent of the data, respectively.)

Dataset / Percentage	DS1	DS2	DS3	DS4	DS5
Small Groups	39%	66%	77%	82%	87%
Medium Groups	51%	28%	19%	15%	11%
Large Groups	10%	6%	4%	3%	2%

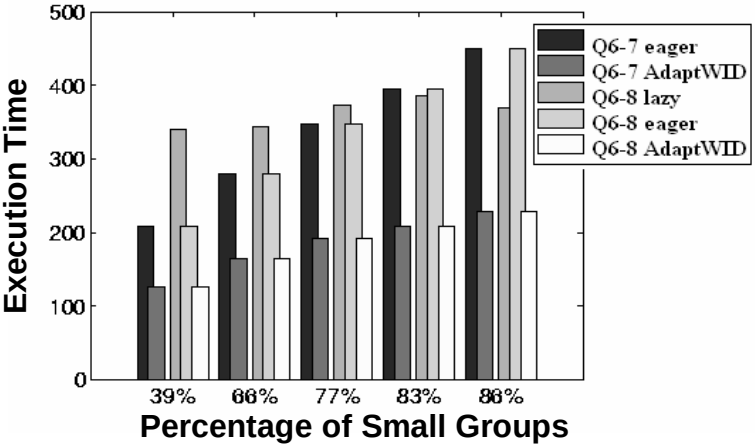
Data Generation: Using network-packet headers from the Passive Measurement and Analysis project [48], we generated input streams for Q6-7 and Q6-8. For Q6-7, we generated an ordered input stream with data-distribution skew. To simulate data distribution skew, we assign data to three types of groups: small, medium, and large. A small group is defined to contain one record; a medium group contains an average of 15 records, and a large group an average of 300 records. To vary the data skew, we distribute 1, 3, 5, 7, or 9 percent of the data to small groups, a fixed 20 percent of the data to medium groups, and the remainder to large groups. Every group is assigned a group-id, and we replaced the srcIP attribute value of the original data with the group-id. The result is five data sets, each with the same number of records, but different record distributions, as shown in Table 6-1. The data set size is approximately 135 MB. For Q6-8, we generated three data streams to emulate three approximately synchronized data sources with each individual stream is skewed in data distribution: two streams simulating the main links with high data volume (approximately 4000 tuples/second) and one stream simulating the control link that contains a small amount of data (almost empty). The total data set size is approximately 135 MB. We varied



(a) Memory performance over data-skewed sources.



(b) Latency performance over data-skewed sources.



(c) Execution itme performance over data-skewed sources.

Figure 6-19 WID vs. AdaptWID for a tumbling-window query over a single data source, Q6-7, and a sliding-window query over three data sources, Q6-8

the amount of time skew between the Control stream and the main streams; there is no time skew between the two main streams.

Experiments and Results: Figure 6-19 compares eager aggregation and AdaptWID on Q6-7, and lazy aggregation, eager aggregation, and AdaptWID on Q6-8. Graphs (a), (b), and (c) in Figures 6-19 show memory usage, latency, and execution time, respectively. Memory usage is the maximum memory used during query execution. Latency is the difference between the arrival time of a punctuation and the output time of the aggregates covered by that punctuation. Execution time reflects the CPU cost, and is the running time of a query over the input data set. The numbers reported in our performance study for latency and execution time are the average of eight runs. As Figures 6-19 shows, AdaptWID outperforms eager aggregation in all three categories for both queries: The memory benefit of AdaptWID is significant, confirming our expectations. The execution time and latency benefit of AdaptWID is due to the hash table in the AdaptWID aggregate operator containing many fewer entries than the hash table used for eager aggregation, greatly reducing the access time. In general, compared to WID, the benefits of AdaptWID increases as the percentage of small groups increase, because with more small groups, WID needs to maintain more partial aggregates while AdaptWID leverages lazy aggregation for the small groups.

Although WID (the eager-aggregation approach) is generally a better implementation for stream query evaluation than the buffering implementation (the lazy-aggregation

approach), it might not be space efficient in dealing with data distribution skew. AdaptWID adapts between the two implementations based on stream properties, including stream volume, arrival order, and synchronization of different data sources, and achieves better performance than both WID and the buffering implementation.

In summary, we presented three order-insensitive implementations of window aggregation: WID, which is directly based on our semantic definition for window aggregation, and two extensions of WID, Paned-WID and AdaptWID, which optimize for execution time and memory usage, respectively. In the rest of the thesis, we will be looking further at disorder-tolerant operator implementations and stream-system architectures.

Chapter 7

ORDER-INSENSITIVE IMPLEMENTATIONS OF WINDOW JOIN

Current window join implementations often require ordered input streams and also need to maintain output order, as current stream query operators normally assume that streams should be ordered. In this chapter, we present order-insensitive implementations of window join. Such implementations do not need to rely on ordered streams for purging state and can output results on the fly without enforcing output order. Thus, these implementations normally have better latency performance than the order-sensitive ones, because input tuples can be processed on the fly without the delay of waiting for late tuples and result tuples can be released on the fly without being sorted. Also, order-insensitive implementations of window join often have a a smaller footprint than the order-sensitive ones because sorting the results of join may require a large amout of memory.

7.1. Order-Insensitive Implementation of Window Join

In the following, we present order-insensitive implementations of sliding-window join and tumbling-window join. These implementations make no restrictions on the arrival order or synchronization of their input. We begin with sliding-window join.

Figure 7-1 shows the OA-Join (*O*rder-*A*gnostic Join) algorithm for sliding-window join. The input streams are S0 and S1, the progressing attribute is *ts*, and the window condition is equivalent to the band predicate, $(S_0.ts - \text{RANGE}_0) \leq S_1.ts \leq (S_0.ts +$

$RANGE_1$). For ease of presentation, we ignore join predicates on other data attributes in the WHERE clause. OA-Join maintains a tuple until it can confirm that no tuples from the other input stream will join with that tuple. OA-Join also maintains the low-watermark timestamp of each input stream. It is important to note that new tuples do not always need to be stored. As the *ProcessTuple*() function shows, if the *ts* value of a new tuple is smaller than the high-watermark bound minus the RANGE value for the other input, that tuple can be processed on the fly and discarded, because all the tuples with which it needs to join have already arrived on the other stream. The amount of state that OA-Join needs to maintain depends on the progress of the input streams. In general, the progress of the left input indicates which tuples from the right input can be purged, and vice versa.

In our algorithm, a join result contains both $S_0.ts$, and $S_1.ts$, the windowing attribute values of the two input streams. This result construction thus allows a subsequent operator to use $S_0.ts$, $S_1.ts$, the pair ($S_0.ts$, $S_1.ts$) or a function of $S_0.ts$ and $S_1.ts$ (e.g., max($S_0.ts$, $S_1.ts$) or min($S_0.ts$, $S_1.ts$)) as its progressing attribute. Some existing window join implementations produce only one timestamp attribute in the join result. This attribute is often equal to one of the two input timestamps; other implementations use the maximum of the two input timestamps as the timestamp of the result. As shown in the *ProducePunctuation*() function, OA-join produces punctuation for $S_0.ts$ and $S_1.ts$ separately, which we term *individual punctuation*. Individual punctuation indicates the progress of the join result on either $S_0.ts$ or $S_1.ts$, and allows subsequent operators to deduce stream progress even when their progressing attribute involves

both $S_0.ts$ and $S_1.ts$ or a function of $S_0.ts$ and $S_1.ts$. For example, if the operator's progressing attribute is max($S_0.ts$, $S_1.ts$), it can progress to *s* when it receives punctuation for *s* from both S_0 and S_1; if its progressing attribute is min ($S_0.ts$, $S_1.ts$), it can progress to *s* when it receives the first punctuation for *s* from either S_0 or S_1. However, as we will explain in the next section, providing the progress of the join result on the combination of $S_0.ts$ and $S_1.ts$ may allow subsequent operators to produce results sooner.

Our order-insensitive implementation of tumbling-window join—equivalent to stream join with an equality predicate on progressing attributes—is similar to that of sliding-window join, but simpler because windows on both input streams have the same size. Figure 7-2 shows the OA-Join implementation for tumbling-window join with predicate, $S_0.ts$/RANGE = $S_1.ts$/RANGE, using integer division. The main difference between OA-Join for sliding-window join and OA-Join for tumbling-window join is in the predicates in the the *ProcessTuple*(), *ProcessPunctuation*() and *ProducePunctuation*() functions, including the predicate that *ProcessTuple*() uses to determine if a tuple should be stored, the predicate that *ProcessPunctuation*() uses to determine if a tuple can be purged, and the predicate that *ProducePunctuation*() uses to determine the output punctuation value.

Discussion: Both order-sensitive implementations of join and our order-insensitive implementation, OA-Join, can produce join results immediately, although different state management may cause differences in output delay. The order-sensitive

implementations of join require ordered input and rely on the ordering to purge state.

```
State Maintain
b0, b1: bounds on the low-watermark of left and right input,
respectively; initialized to –∞;
M0, M1: sets of tuples maintained on left and right input,
respectively; initialized to ∅;

Join(x)
let Si be the input stream to which x belongs;
if x is a tuple
        ProcessTuple(x, Si);
else if x is a punctuation
        ProcessPunctuation(x, Si);

ProcessTuple(t, Si)
join t with matching tuples in M1-i;
if t.ts ≥ b1-i – RANGEi
   add t to Mi;

ProcessPunctuation(p, Si)
bi = p.ts;
for each k in M1-i
   if k.ts < p.ts  – RANGE1-i
        purge k;
ProducePunctuation (p, Si);

ProducePunctuation(p, Si)
output a punctuation for S1-i.ts with value min(bi –RANGE1-i,  b1-i);
output a punctuation for Si.ts with value min(b1-i – RANGEi,  bi);
```

Figure 7-1 OA-Join for sliding-window join.

OA-Join purges state based on punctuations. The amount of state that the order-sensitive implementations of join and OA-Join maintain internally is similar. The OA-Join implementation may require maintaining even more internal state than the order-insensitive implementations, because input-stream disorder delays the expiration of

tuples. However, the order-sensitive implementations may need to buffer output tuples to maintain order for sliding-window join, while OA-Join can release output tuples in any order, and requires no output buffer.

State Maintained:
b_0, b_1: bounds on the low-watermark of left and right input, respectively; initialized to $-\infty$;
M_0, M_1: sets of tuples maintained on left and right input, respectively; initialized to $\varnothing$;

Join(*x*)
let S_i be the input stream to which *x* belongs;
if *x* is a tuple
    ProcessTuple(*x*, S_i);
else if *x* is a punctuation
    ProcessPunctuation(*x*, S_i);

ProcessTuple(*t*, S_i)
join *t* with matching tuples in M_{1-i};
if $t.ts \geq b_{1-i}$/RANGE
  add *t* to M_i;

ProcessPunctuation(*p*, S_i)
$b_i = p.ts$;
for each *k* in M_{1-i}
  if $k.ts < p.ts$ / $RANGE_{1-i}$
    purge *k*;
ProducePunctuation (*p*, S_i);

ProducePunctuation(*p*, S_i)
output a punctuation for $S_{1-i}.ts$ with value min($\lceil b_i/\text{RANGE} \rceil$*RANGE, b_{1-i});
output a punctuation for $S_i.ts$ with value min(b_i, $\lceil b_{1-i}/\text{RANGE} \rceil$*RANGE);

Figure 7-2 OA-Join for tumbling-window Join.

7.2. Producing Finer-Granularity Punctuation

Although subsequent operators can deduce the progress of the join result based on *individual punctuation* of the two timestamps in result tuples, the OA-Join implementations can also produce another form of punctuation on the timestamp attributes of both S_0 and S_1, which we term *joint punctuation*. As we discuss below, joint punctuation may improve the latency of subsequent operators. Consider the following query, Q7-1, which counts established TCP connections per time period in the network traffic between two links, S_0 and S_1. It defines a band predicate ($S_0.ts - 2$) $\leq S_1.ts \leq (S_0.ts + 2)$—the band is symmetric and thus it is equivalent to ($S_1.ts - 2$) $\leq S_0.ts \leq (S_1.ts + 2)$. We will refer to the predicate as "the band" for this discussion. The band is used to set a practical constraint on the range of packets that each packet may need to be matched with. Thus, Q7-1 joins SYN and SYN_ACK packets from S_0 and S_1 for a network connection between S_0 and S_1 over the past 2 mintues of each other, and computes the number of SYN and SYN_ACK pairs for each corresponding pair of timestamps ($S_0.ts$, $S_1.ts$).

Q7-1: "Count the number of SYN, SYN_ACK pairs (SYN_ACK arrives after SYN for no more than 2 minutes) for network connections between S_0 and S_1 for each time period, i.e., each pair of timestamps, ($S_0.ts$, $S_1.ts$)."

```
SELECT S0.ts, S1.ts, count(*)
FROM   S0 [WA ts, RANGE 2 min],
       S1 [WA ts, RANGE 2 min]
WHERE      S0.srcIP = S1.destIP and S0.destIP = S1.srcIP and
           S0.srcPort = S1.destPort and S0.destPort = S1.srcPort and
           ((S0.ts < S1.ts and S0.flag = SYN and S1.flag = SYN_ACK) or
           (S0.ts > S1.ts and S0.flag = SYN_ACK and S1.flag = SYN))
GROUP BY S0.ts, S1.ts;
```

Here, joint punctuation is punctuation on $S_0.ts$ and $S_1.ts$ together, and can allow the count to output results with less delay than with individual punctuation. Figure 7-3 and Figure 7-4 provide the pseudo-code for producing joint punctuation for sliding-window join and tumbling-window join, respectively. These algorithms only produce joint punctuation with $S_0.ts$ and $S_1.ts$ values within the join window of each other, as those are the only result *ts* pairs that the join may produce. Unlike individual punctuation, joint punctuation production is independent of window size, and therefore can be produced earlier than individual punctuation. This difference may be significant for joins with a large window size.

```
ProducePunctua(p, S_i)
b_i = p.ts;
if b_i – b_1-i ≤ RANGE_1-i or b_1-i – b_i ≤ RANGE_i
    output a punctuation with values b_i and b_1-i for S_i and S_1-i, respectively;
```

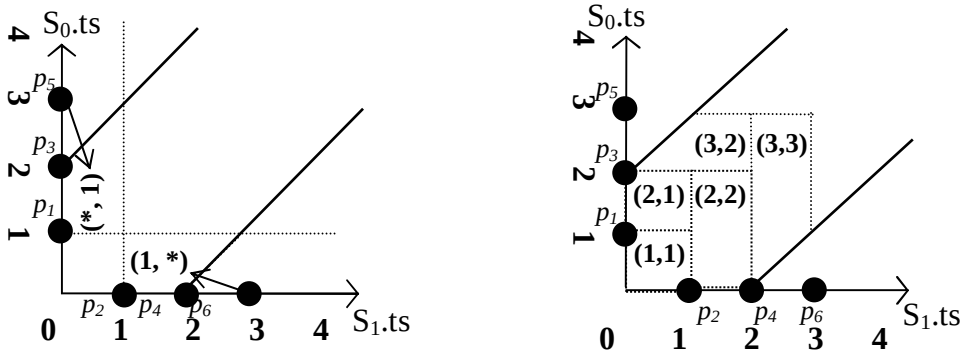
Figure 7-3 Joint punctuation production for sliding-window OA-Join.

```
ProducePunctuation (p, S_i)
b_i = p.ts;
if b_i/RANGE = b_1-i/RANGE_1-i
    output a punctuation with values b_i and b_1-i for S_i and S_1-i, respectively;
```

Figure 7-4 Joint punctuation production for tumbling-window OA-Join.

Figures 7-5 (a) and (b) illustrate the progress information that individual punctuation and joint punctuation, respectively, can provide for the count operator in Q7-1. In Figure 7-5, the x- and y-axes indicate the *ts* values of S_0 and S_1, respectively; the solid

lines indicate the region of timestamps that satisfy the band predicate. Dark dots on the axes represent punctuation in the input streams. The indices of input punctuation represent global arrival order. Number pairs represent joint output punctuation on S_0.*ts* and S_1.*ts*, and dotted lines outline the coverage of each output punctuation.



(a) Individual **(b) Joint**

Figure 7-5 Individual vs. joint punctuation – produced by Q7-1

Observe that joint punctuation allows the Count operator in Q7-1 to output results with less delay. Consider the aggregate group in the Count operator with S_1.*ts* value equaling 1 and S_0.*ts* equaling 0. With individual punctuation, the Count operator can output this group when it receives punctuation (*, 1); with joint punctuation, it can output the group when it receives punctuation (2, 1). Punctuation (*, 1), which indicates that join has produced all results with S_1.*ts* value smaller than 1, is produced by join when it receives punctuation *p*5—a punctuation *p* on S_i allows the output of punctuation on S_{1-i} with timestamp value *p.ts* less the window size. Punctuation (2, 1), which indicates that join has produced all results with S_0.*ts* value smaller than 2 and S_1.*ts* value smaller than 1, is produced when join receives punctuation *p*3. In our

example, *p*3 is received one minute before *p*5. Thus, with joint punctuation, the Count operator outputs the group one minute earlier than with individual punctuation.

In general, an individual punctuation is defined on a single timestamp attribute and covers a "slab" region as shown in Figure 7-5(a), while a joint punctuation is defined on both timestamp attributes and covers a "box" region as shown in Figure 7-5(b). A "slab" finishes when all the tuples in the covered region have been produced and thus it finishes later than most of individual boxes covering the same region.

7.3. Performance Study of OA-Join

We compared the OA-Join algorithm and order-preserving implementations for sliding-window join that guarantee the order of the results of join using NiagaraST. The experiments were conducted on a Dual-Core AMD Opteron™ Processor 2214 with 4GB main memory, running Ubuntu Linux 2.6.17-10-server, and Sun® Java VM 1.5.

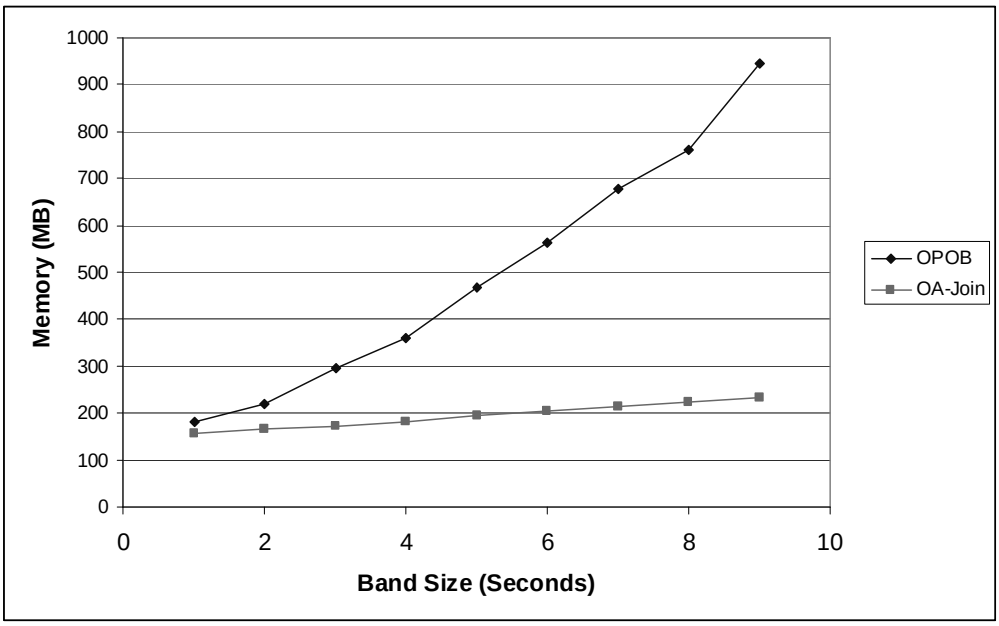
Data Generation: For our experiments, we generated data streams using network-packet headers from the Passive Measurement and Analysis project [48]. We generated two data streams, with data volume approximately 4000 tuples/second, called M1 and M2. The total data set size is approximately 135 MB. In our experiments, M1 and M2 are ordered and synchronized.

Experiment 1: The first set of experiments compare the memory, execution time and latency performance of OA-Join and an order-preserving, output-buffered implementation, which we call OPOB-Join (*O*rder-*P*reserving *O*utput-*B*uffered Join),

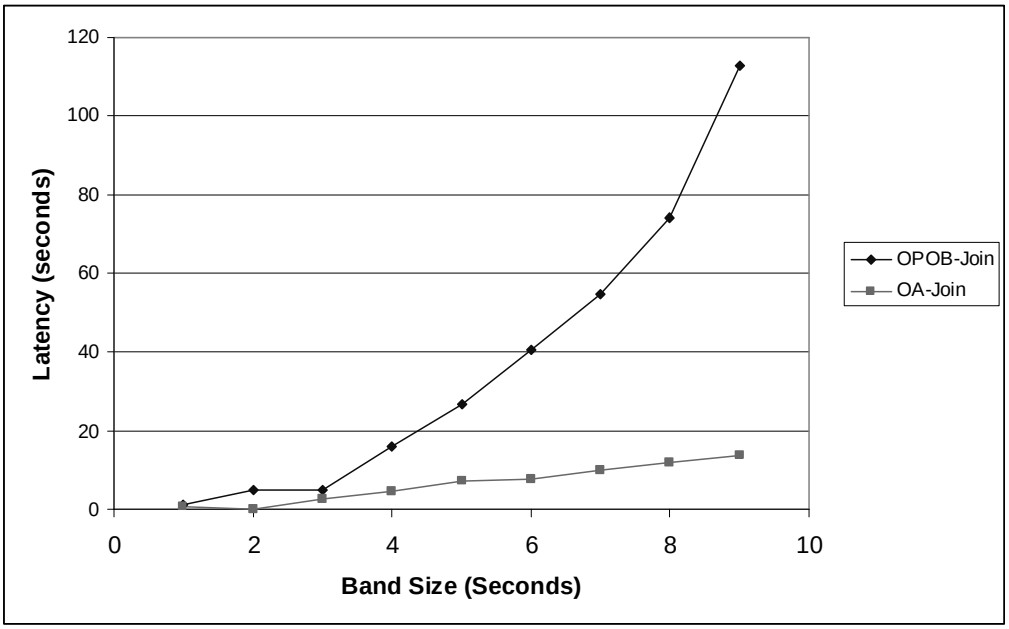
of a sliding-window join query for Q7-2 below in NiagaraST. Q7-2 joins packets that satisfy the window condition and are also in the same NetFlow, but for different directions, and requires the output of join to be ordered on the timstamp of the first input stream. The OPOB-Join implementation of sliding-window join does not output the join results on the fly; instead, it uses an output tuple buffer to sort the results and output them in order. We varied the window size *n* of the join operator from 1 second to 9 seconds, and measured the maximum memory usage, latency and execution time of the query.

Q7-2: "Count the number of network packet pairs in each minute from M1 and M2, in the same Netflow but in the opposite direction."

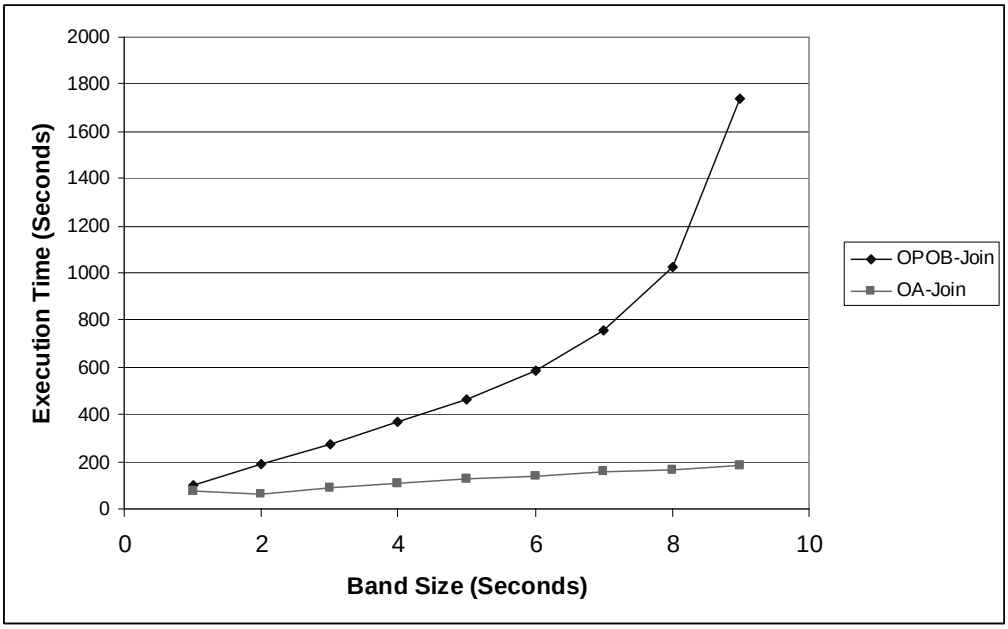
```
SELECT count(*) [RANGE 1 minute, SLIDE 1 minute, WA M1.ts]
FROM   M1 [RANGE n, WA ts],
       M2 [RANGE n, WA ts]
WHERE M1.srcIP = M2.destIP and M1.destIP and M2.srcIP and
       M1.srcPort = M2.destPort and M1.destPort = M2.srcPort
```



(a)



(b)



(c)

Figure 7-6 Memory, latency and execution time comparison of OA-Join and OPOB-Join implementation for a sliding-window join query, Q7-2, for different band sizes

Figures 7-7 shows (a) maximum memory usage, (b) latency, and (c) execution time comparisons of the OA-Join and the OPOB-Join implementation of Q7-2. The y-axes of (a), (b), and (c) show maximum memory usage, median latency, and execution time, respectively. Latency is the difference between the output time of an aggregate and the arrival time of punctuation from the input streams that triggers the output of the aggregate. Execution time reflects the CPU cost, and is the elapsed time of a query running at full speed over the input data set. The latency and execution-time numbers are the average of 8 runs. The memory overhead is deterministic for a given input order and is the same across runs. OA-Join significantly outperforms the OPOB-Join

implementation on Q7-2, especially on memory and latency, as OA-Join can avoid sorting the output of the join before aggregating.

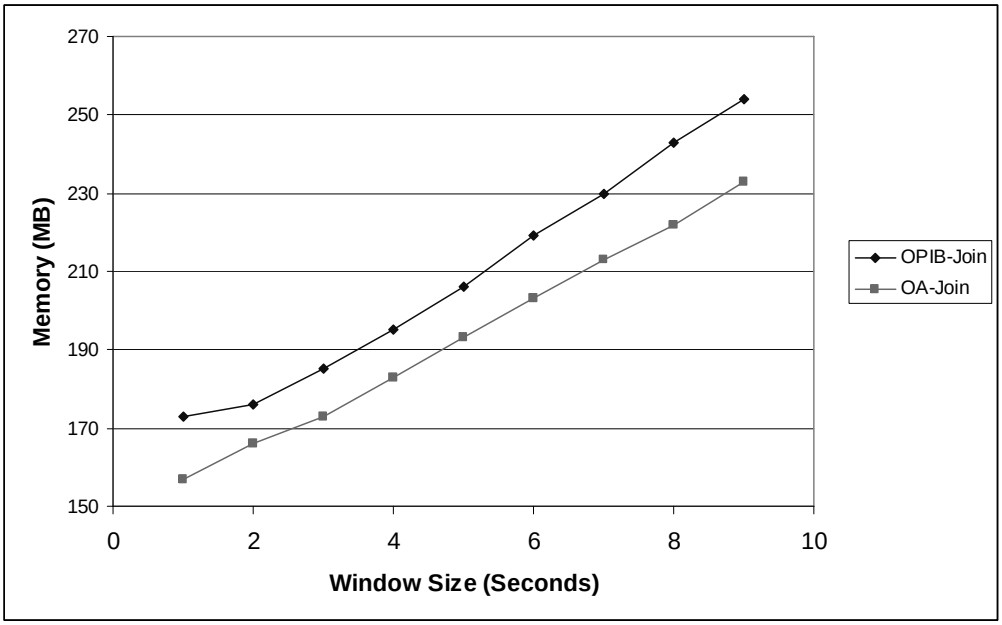


Figure 7-7 Memory comparison of OA-Join and the OPIB-Join implementation for a sliding-window join query, Q7-2, for different band sizes

Experiment 2: In this experiment, we compared the memory uage of OA-Join and another order-preserving implementation of sliding-window join, which we call OPIB-Join (*O*rder-*P*reserving, *I*nput-*B*uffered Join). This OPIB-Join implementation performs join in such an order that the results are produced in the desired output order, and thus it may have better memory performance than the OPOB-Join implementation for join queries when join predicate is not very selective—that is, when the output data volume of the join is higher than its input data volume. However, the input-buffered implementation incurs more delay in producing join results and thus may have higher latency than the output-buffered implementation. In this experiment, we use the same

query, Q7-2, and the same two data streams, M1 and M2, as in Experiment 1. Figure 7-8 shows the memory comparision of the OA-Join and the OPIB-Join of Q7-2. OA-Join uses slightly less memory than the OPIB-Join implementation for all window sizes in the experiment. Thus, although OPIB-Join uses much less memory than OPOB-Join, OA-Join is still better than OPIB-Join in memory usage for Q7-2.

In summary, we presented order-insensitive implementations for slidng-window join and tumbling-window join in this chapter. These order-insensitive implementations benefit from avoiding the overhead of maintaining output order, but need query operators following them to be order-insensitive. For example, if a sliding-window join is followed by a window aggregation, order-insensitive implementations of window aggregation are required for the join to use an order-insensitive implementation. Note that the ability of window aggregation to handle disordered input leads to performance benefits of window join. The next chapter extends this notion of "disorder-handling benefits" systematically to whole queries.

Chapter 8

OUT-OF-ORDER STREAM QUERY EVALUATION

In this chapter, we present a new, order-insensitive stream query processing architecture, *OOP* (*O*ut-of-*O*rder *P*rocessing), which is motivated by the order-insensitive query operator implementations presented in previous chapters. The OOP architecture takes our idea of separating stream progress and physical stream arrival one step further and enables out-of-order processing at the system level. In an OOP architecture system, punctuation is inserted into input streams and query operators are required to propagate punctuation so that each stream query operator receives stream-progress information from its input(s). Thus, with the OOP architecture, query operators in a stream-query execution plan can be order-insensitive.

Compared to the previous IOP (*I*n-*O*rder *P*rocessing) approach that requires maintaining stream order, the benefits of the OOP architecture include reduced memory usage and response time. In addition, for massive data streams such as network traffic from the backbone network of AT&T, OOP also leads to better workload shaping and thus increases the maximum data rate that a query can support without dropping tuples. The benefits of the OOP architecture come from avoiding the need to enforce order on streams, especially inter-operator streams. As we will discuss later in this chapter, even when input streams are ordered, inter-operator streams can be disordered. More importantly, in real-world applications, stream-processing

systems are often deployed in distributed computing environments, where the cost of enforcing order on inter-operator streams may be prohibitive.

In contrast to existing techniques for handling disorder, such as slack, we argue that OOP provides system-level support in propagating stream progress and is thus more effective and efficient in dealing with disorder. Recall that slack is a parameter of a query operator specifying an amount of delay for waiting for delayed tuples, for example, 10 tuples or 1 minute, and handles disorder at the query-operator level. A query operator with slack will retain each tuple in a buffer for the specified delay period, attemping to put delayed tuples into order. The user or stream system needs to provide the slack parameter for each query operator, but setting the slack parameter is a non-trivial problem, as we explore in the next example.

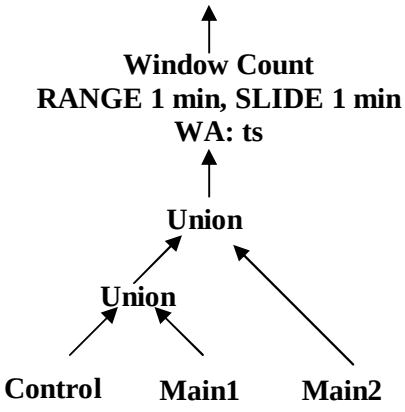


Figure 8-1 Query plan for query Q1-1 in Chapter 1

Let us consider the query Q1-1 from Chapter 1 again and compare the OOP and IOP (with slack) evaluation of this query. The logical plan of Q1-1 is shown in Figure 8-1 again. Q1-1 computes the count of packets over the combination of three streams.

When there are time skews among the three streams, the combined stream is disorded. If the Window Count operator handles disorder using slack, the Union operators can pass tuples through immediately. Then, however, the system must determine the slack parameter for the Window Count operator. Unless the time skew of the input streams is known and fixed, it is very difficult to set slack for the Window Count operator so it precisely captures the disorder of the combined streams. Input tuples will be dropped if the slack is set too small, while a latency penalty will be incurred if the slack is set too large. Further, even when the time skew of the input streams is known and fixed, setting slack for query operators processing intermediate streams is non-trivial. For example, if we replace the Union below the aggregate with a sliding-window join, the disorder in the output of the join (in terms of both tuple count and the amount of maximum delay on the progressing attribute) will be greater than that of the input streams; hence the window aggregation will need to use a larger slack than that for the union of input streams. To our knowledge, no one has presented a comprehensive method for calculating the appropriate slack on an operator's output stream from the slack of its input streams. Further, the operator producing the intermediate stream, such as a sliding-window join operator, may have the exact progress information that its downstream operator requires, and thus it is wasteful to have the downstream operator re-discover or estimate it.

OOP deals with disorder by requiring query operators to propagate punctuation that communicates stream progress. In an OOP system, each query operator receives punctuation and thus does not need to deduce stream progress from observations of its

input(s). It is important to note that the main difference of OOP and IOP is the means they use to communicate stream progress. Even with slack, stream-query operators in IOP systems still need to deduce stream progress from stream arrival order, while in OOP systems, stream progress is explicitly provided to query operators.

In this chapter, we will start with the generation of punctuation. Then, we briefly discuss the order-insensitive implementations of stream query operators—to go along with our previous order-insensitive implementations for window aggregation and window join. We will discuss other operators including Input, Select, Apply, Project, Duplicate Elimination, and Union. We also discuss the benefits of OOP, including benefits for aggregation queries, join queries, and workload smoothing. Finally, we present experiments comparing OOP versus IOP in Gigascope and NiagaraST.

8.1. Punctuation Generation

In this work, we use punctuation to carry stream progress information. Note that although we choose punctuation—a data-driven mechanism—to propagate stream progress in this thesis, OOP can also work with other non-data-driven stream progress mechanisms, such as operators periodically polling their input operators for progress bounds, or having a global scheduler track operator progress.

In general, any information that IOP systems use to ensure order on input streams can be used to detect or bound the progress of those streams, for example, knowledge that an input stream is ordered, or limitations on the amount of delay expected or allowed. Recall that we assume that streams must have a progressing attribute and the low-

watermark of the progressing attribute value indicates the progress of the stream. Also, punctuation typically is defined on the progressing attribute of a stream (i.e., the punctuating attribute is the progressing attribute). Here are a few examples of how stream low-watermark can be detected and thus how punctuation can be created and inserted into a stream.

- If an input stream is known a priori to be ordered, the low-watermark after a prefix of the stream is the progressing attribute value of the most-recently-arrived tuple.
- If an input stream contains out-of-order tuples but it is known that a tuple will not be delayed by more than *n* tuples, IOP can enforce stream order by buffering and re-ordering the input stream, as in the BSort operator of Aurora [2]. In this case, the low-watermark of the input stream can be estimated by the maximum progressing-attribute value of arrived tuples excluding the last *n* tuples.
- Widom et al. [61] propose a heartbeat mechanism to enforce order on out-of-order input caused by time skew and transmission delay, and propose algorithms that estimate parameters that characterize the possible sources of disorder and generate heartbeats based on these parameters. Such algorithms can be used to estimate stream progress and then generate punctuation in similar situations.

In order to adapt an IOP system to OOP, we must either add punctuation to the system, or, if the system already supports punctuation, we must extend it to fully

support out-of-order processing. (Some existing IOP systems, such as Gigascope, support punctuation for handling lulls.)

In Gigascope, punctuation is initiated by timer callbacks. Assuming an input stream is ordered, the callback function can insert a punctuation carrying the largest progressing-attribute value observed so far in the stream every time the timer fires. However, during lulls, the observed *data time*—the current value of the progressing attribute—drifts away from the system time. When the difference between the data time and system time is above a predefined threshold, *s*, the callback function inserts punctuation to advance the data time to (current system time – *s*).

One must be careful when adding punctuation to IOP systems, especially for stream systems that support batch processing (i.e., query operators are invoked for a "batch" of input tuples instead of for each individual tuple). Punctuation may trigger ouput (e.g., for aggregate queries) or may be used to purge state (e.g., for join queries). Thus, for stream systems that support batch processing, punctuation should be treated as a high-priority tuple: Once a punctuation arrives, the in-progress batch should be considered complete and should be shipped to down-stream operators. Note that this completion of a batch affects only the timing of tuple transmission and does not affect result values. Punctuation delayed by batch processing may delay result production and thus increase latency, particularly for sparse streams.

IOP systems that already support punctuation may require non-trivial effort to extend punctuation to fully support OOP. First, OOP systems rely on punctuation to make progress, thus the system should produce punctuation at a granularity finer than both

the smallest window size and the smallest window slide allowed in the stream system. The granularity of punctuation used for handling lulls in IOP systems can be much coarser, as such punctuation only needs to guarantee that stream queries make progress during lulls. Second, timer callbacks for generating punctuation may initiate duplicate punctuation, if the timer is set at a granularity fine enough to satisfy the smallest window slide. For efficiency, it is desirable to avoid such duplicates; further, it is also desirable to produce only punctuation that matches the boundaries of the smallest window slide currently used in the system. For example, if the smallest window slide used by queries currently running in the system is 5 seconds, it is desirable to produce punctuation with a 5-second granularity and no finer. Third, to provide stream-progress information efficiently, a query operator should choose what punctuation to produce based on the requirements of the operator that consumes its result. Tucker [71] has proposed a Describe operator that provides punctuation appropriate for downstream operators. The describe operator filters out punctuation that will not help downstream operators and rolls incoming punctuation up to the appropriate level.

We have experimented with punctuation in two systems, Gigascope and NiagaraST. Gigascope supports timer-driven punctuation [33]—in a low-level sub-query, a timer callback function fires every second (in wall-clock time), and a punctuation carrying the stream low-watermark is inserted into the input stream. As the input stream to the low-level sub-query is ordered, determining the punctuation value is straightforward. NiagaraST supports data-driven punctuations. In the absence of external punctuation

provided by a data source, NiagaraST can insert punctuation into the data stream. In a simple scheme, if a data stream is known to be ordered, NiagaraST inserts punctuation into the stream when it observes that the value of the progressing attribute has changed by a predefined amount.

8.2. Order-Insensitive Implementation of Query Operators

In this section, we briefly discuss the order-insensitive implementation of stream query operators (beyond window aggregation and window join) and compare them to their order-sensitive counterparts. Order-sensitive imiplementations typically require ordered streams and need to preserve stream order; order-insensitive implemenations do not. Compared to order-sensitive implementations, order-insensitive implementations free query operators from the burden of preserving stream order and thus often improve in the operators' memory and latency performance.

Input: The Input operator is the interface between external data streams and other query operators in a stream system. The implementation of the Input operator can be very application-dependent. Order-sensitive implementations of the Input operator need to guarantee stream order; order-insensitive implementations of the Input operator need to put punctuation into the input stream. Both implementations need the the same type of information about data arrival, either to ensure stream order or to insert punctuation. For example, both can benefit from knowing whether the stream is ordered or the maximum amount of disorder in streams.

Select, Apply, Project: As most unary query operators are neither blocking nor stateful, they do not require ordered streams to process tuples; also, the pipelined implementations used in regular relational DBMSs of these operators work for stream systems. If the input stream is ordered, the pipelined implementations naturally preserve order. For unary operators, the progress of the input stream directly determines the progress of the output stream, and thus punctuation processing for such operators is simple. Select passes through punctuations. Apply and Project need to first transform input punctuation into the output schema before putting it to the output. For the Project operator, we assume its output includes the progressing attribute(s).

DupElim: Duplicate elimination (DupElim) also naturally preserves order; the issue is when state can be purged. The order-sensitive implementations of DupElim can remove state whenever the progressing attribute advances, and the order-insensitive implementation relies on punctuation to purge its state.

Union: The order-preserving Merge operator, as in Gigascope [33], is an order-sensitive implementation of Union. The Merge operator must buffer tuples from one input during a lull or delay on the other input in order to assure ordered output. Punctuation can be used to reduce the buffering required due to lulls on one input, but if there is time skew between the inputs, the Merge operator must still buffer the earlier input. The memory and latency costs of Merge are determined by the lulls (or punctuation granularity when punctuations are available) and offsets between the input streams. (Recall that we define offsets between different streams in Section 6.3.)

The order-insensitive implementation of Union, which we call Meld, can pass input tuples through immediately. The Meld operator needs to buffer input punctuations in order to correctly produce output punctuations, but punctuations typically constitute only a small fraction of stream volume. Further, for linear punctuation, when punctuations are guaranteed to arrive in the desired order, the order-insensitive implementation needs only remember the most recent punctuation on each input. For Meld, we require that the progressing attributes of both input streams are the same. Suppose that the input streams are R and S, and the value of the last punctuation arrived on them are R.*punctVal* and S.*punctVal*, respectively. When a new punctuation *p* with value *ts* arrives in R, the Meld operator can output a punctuation with value min(*ts*, S.*punctVal*), and vice versa for a punctuation from S. An issue here is that the Union operator may produce duplicated punctuations. For example, suppose that R.*punctVal* and S.*punctVal* are 20 and 19 respectively. If punctuation from S for times 20, 21 and 22 arrives before any further punctuation from R, the union will output punctuation for 20 at least three times. To avoid producing duplicate punctuation, the Meld operator can maintain the value of the last punctuation output, and only output a punctuation if its value is greater than the value of the last punctuation.

State Maintained:
b_0, b_1: bounds on the low-watermark of left and right input, respectively; initialized to $-\infty$;
o: low-watermark of the output stream; initialized to $-\infty$;

Union(x)
let S_i be the input stream to which x belongs;
if x is a tuple
    ProcessTuple(x, S_i);
else if x is a punctuation
    ProcessPunctuation(x, S_i);

ProcessTuple(t, S_i)
output t;

ProcessPunctuation(p, S_i)
$b_i = p.ts$;
if $o < \min(b_i, b_{1-i})$
    output a punctuation with value $\min(b_i, b_{1-i})$;
    $o = \min(b_i, b_{1-i})$;

Figure 8-2 Order-insensitive implementation of Union—Meld.

The implementation of Meld for linear punctuation is shown in Figure 8-2—the *Union*() function is called for each tuple or punctuation. As compared to the order-enforcing Merge operator implementation, the Meld implementation is lightweight in terms of both memory and latency. The only state that the Meld operator implementation maintains is the most recent punctuation value from each input stream and for the output stream. Group-wise punctuation may require maintaining such state for each group. Meld passes tuples through immediately, and it emits punctuation with the minimum progressing-attribute value observed from both streams (minus duplicates). Since the Union operator is necessary for stream queries monitoring data

from multiple sources, such as multiple network-traffic links, the lightweight implementation can be a great advantage. When an order-preserving Union is used, both memory and delay incurred by Union can be prohibitive during lulls or in the presence of time skew.

8.3. Cases for OOP

In this section, we discuss the benefits of OOP for aggregation and join queries, as well as for workload smoothing when processing massive data streams, which can promote higher throughput. As the following examples will illustrate, the benefits of OOP often come from processing (disordered) intermediate streams more efficiently.

8.3.1. OOP Benefits for Aggregation

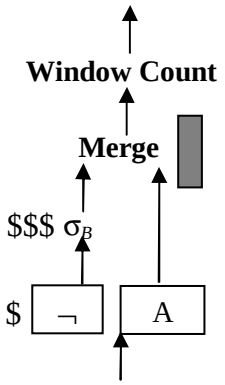


Figure 8-3 Merge enforces order on intermediate results even when the query has a single, ordered input stream

In OOP stream systems, as out-of-order tuples are handled without delay, aggregation queries may have a smaller footprint and better latency compared to IOP systems.

Even for queries with a single, ordered input stream, disorder may occur in intermediate streams. For example, the input stream may be split and processed through different sub-queries (such as might be needed for network-protocol simulation), and the combination of the sub-query results may be disordered. Figure 8-3 shows an example with an ordered input stream: The input is split according to an inexpensive predicate A; tuples not satisfying A are put through an expensive predicate B before being merged with the stream of tuples satisfying A. The output of the merge contains tuples that satisfy either A or B; this result is fed to a Window Count operator.

With the non-OOP alternatives, either the Union operator needs to enforce order on data—with a cost of memory and latency—or the Window Count operator has to use slack to account for the disorder caused by the delay from the expensive predicate B.

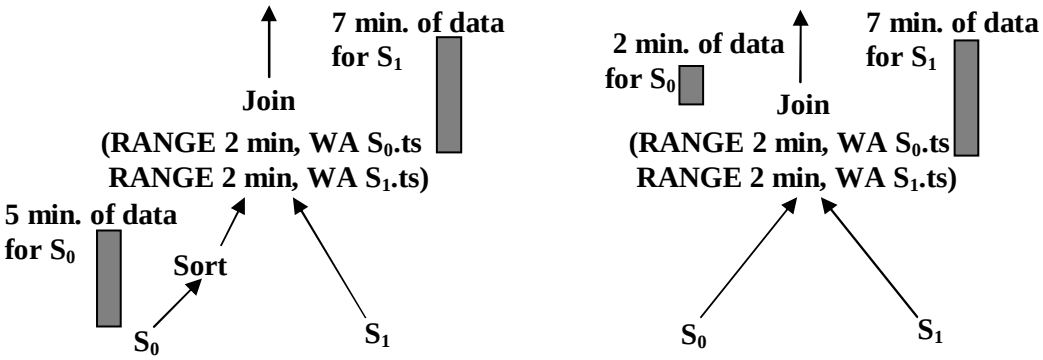
With OOP, the Union operator passes tuples through immediately, and every query operator propagates punctuation; thus, Window Count receives accurate stream progress information. Also, using WID, tuples can be immediately reduced into partial aggregates by the Window Count operator. Overall, maintaining partial aggregates is much less space intensive than buffering tuples and keeps tuple-processing delay minimal; propagating stream progress precisely captures intermediate-stream disorder.

8.3.2. OOP Benefits for Join

In OOP systems, the Join operator may often have a smaller footprint and is able to produce results with less delay, as OOP processes each tuple at the earliest possible

time without waiting for late tuples so as to process tuples in order. In particular, the Join operator may process and also purge on-time tuples at the earliest possible moment, thus reducing latency and memory usage.

Consider the join query with input streams S_0 and S_1, and a sliding window, [RANGE 2 minutes, WA *ts*], on each input stream. Assume S_0 may potentially contain a small fraction of tuples that are delayed by at most 5 minutes, and input stream S_1 arrives ordered. S_0 and S_1 are approximately synchronized, which means that—ignoring delayed tuples—tuples from S_0 and S_1 with the same *ts* value arrive at about the same time. Figures 8-3(a) and 8-3(b) show the IOP and OOP evaluations of the sliding-window join query.



(a). IOP (b). OOP

Figure 8-4 Evaluation of a band join (maximum allowed delay in S0 is 5 minutes)

With IOP, due to potentially delayed tuples in S_0, a buffered Sort operator is required to enforce tuple order for S_0. It holds 5 minutes of S_0 tuples, and thus S_0 tuples are generally delayed for 5 minutes. The Join operator maintains 7 minutes of S_1 tuples (2

minutes due to the window condition and 5 minutes due to the delayed S_0 tuples). Join will not need to maintain any state for S_0, as S_0 tuples arrive 5 minutes behind S_1 tuples and hence all matching S_1 tuples are available when each S_0 tuple arrives.

With OOP, both S_0 and S_1 tuples are presented to the Join operator without delay. As Figure 8-4(b) shows, the join maintains 5 minutes of S_1 tuples and 2 minutes of S_0 tuples, because S_0 tuples are purged by S_1 punctuation on time while S_1 tuples are purged late due to delayed S_0 tuples and punctuation. Overall, the OOP evaluation of the join query maintains 3 minutes less of S_0 tuples, and can produce most join results earlier than the IOP evaluation.

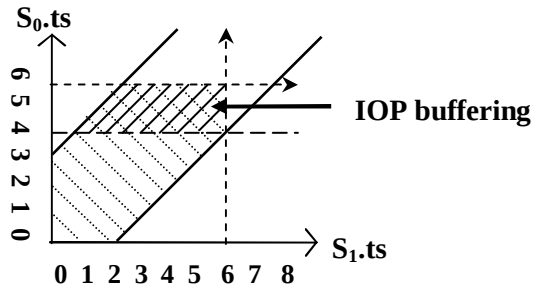


Figure 8-5 Output buffering in IOP band join with output ordered on S0.ts

Further, in OOP systems, the Join operator need not enforce order on its result. In contrast, the IOP approach may a require large amount of buffer space to order the output of a sliding-window join and thus it often is inferior to the OOP approach. Figure 8-5 illustrates this buffer requirement for a join with a sliding window [RANGE 3 minutes, WA *ts*] on S_0, and a sliding window [RANGE 2 minutes, WA *ts*] on S_1. It also assumes that input streams S_0 and S_1 are approximately synchronized,

and assumes that the join results need to be ordered on $S_0.ts$. The single-hatched area illustrates the amount of results produced by the sliding-window join; and the cross-hatched area illustrates the amount of buffering required to order the output. As the figure shows, when both S_0 and S_1 progress to time 6, the join needs to buffer results produced by S_0 tuples with *ts* values between 4 and 6. In general, the required buffering for ordering join output in IOP systems increases with the window size of the Join operator. (The exact amount of buffering is determined by the desired output order, the window conditions, the data rate of the input streams, and the arrival time skew of the input streams.) In OOP systems, results of Join can be released on the fly, without any delay or buffering, and immediately processed by a subsequent operator.

8.3.3. Workload Smoothing

Workload smoothing is critical for systems dealing with massive streams in (near) real time. For such systems, a workload surge at a given operator may overload the system, delay further data processing, and lead to loss of input data or obsolete query results. In this section, we discuss our experiments on workload smoothing with OOP in the Gigascope system. Workload surge can occur either in input or intermediate streams. Workload surges in input streams are often caused by input data bursts, and workload surges in intermediate streams are often caused by blocking operators that are periodically unblocked. For example, when a window ends, window aggregation needs to scan the hash table of partial aggregates to produce results and purge completed items, and outer-join needs to locate and output tuples that were not

matched. Both can lead to a surge in output rate at a window boundary. Here we focus on smoothing intermediate workload surges created by the unblocking of blocking operators.

In the following, we first review the implementation of window-aggregation in Gigascope and how it relates to workload surges. Then, we present a workload-smoothing mechanism, slow-flush, originally implemented in the IOP version of Gigascope for window aggregation and outer-join [33]. Similar workload-smoothing mechanisms are also used in other network-traffic-monitoring systems [36]. Finally, we discuss workload smoothing in the OOP version of Gigscope, using two approaches, slow-flush and lazy-flush. We present both slow-flush and lazy-flush only in the context of window aggregation, but similar techniques also work for outer join.

Aggregation in Gigascope: Gigascope has a two-level architecture typical for high-performance, potentially distributed data-monitoring systems [14], where the low level is used for data reduction and must be lightweight, and the high level is intended for more complex processing. A low-level sub-query processes network packets from a fixed-size ring buffer. Low-level and high-level queries run in different processes (possibly on different machines). Gigascope supports only tumbling-window aggregation natively. An aggregation query is split into a low-level sub-aggregation and a high-level aggregation that rolls up the results of the sub-aggregation. For example, a count query is split into a low-level count query and a high-level sum query. To ensure the low-level sub-aggregation is fast, it uses a fixed-size hash table to maintain aggregates of different groups, so there is no dynamic space allocation. On

hash-table collision, the existing aggregate in the hash table is output to accommodate the new aggregate. At the end of a window, the low-level query flushes the hash table and outputs all aggregates in it. However, if the number of groups is large, flushing the hash table causes a workload surge, during which time the ring buffer can overwrite itself and packets are lost.

Slow-flush mechanism: Gigascope uses a slow-flush mechanism to smooth workload surges at window boundaries in low-level aggregation. With slow-flush, when a window completes, the low-level sub-query gradually outputs aggregates from the previous window while processing new packets, instead of flushing all aggregates from the hash table at once. Figure 8-6 shows the outline of low-level aggregation with slow-flush in the IOP case—Figure 8-6(a) shows how a tuple is processed in a low-level aggregation and Figure 8-6(b) shows the *SlowFlush*() function that is called by the low-level aggregation.

State Maintained:

hashtable: the fixed-size hashtable that low-level aggregation maintains; each hashtable entry represents a group and consists of the grouping attribute values of the group and the partial aggregate for it;

status: a table that records the type of the content for each hashtable entry, including new, old, or empty

Init():

flush_finished = true;

flush_pos = 0;

ProcessTuple(*t*):

if *t* indicates the start of a new window extent

  if (!*flush_finished*)

    flush all the remaining hashtable entries marked as old, and mark them as empty;

    flush_finished = false;

    flush_pos = 0;

if (!*flush_finished*)

  Slow_Flush();

key = hash key of *t*;

if status[key] == empty

  create a new aggregate with *t* in hashtable[key];

if status[key] == new

  if *t* belongs to the group of the exising aggregate

    update the existing aggregate with *t*

  else

    flush all the hash entries in hashtable marked as old;

    output the existing aggregate in hashtable[key];

    create a new aggregate with *t* in hashtable[key];

if status[key] == old

  flush hashtable[key];

  create a new aggregate for *t* in hashtable[key];

(a): the outline

```
SlowFlush()
if (!flush_finished and status[flush_pos] == old)
    output hashtable[flush_pos];
    status[flush_pos] = empty;
    flush_pos++;
    if (flush_pos > hashtable.size)
        flush_finished = true;
```

(b) the SlowFlush() function

Figure 8-6 Low-level aggregation with slow flush—the *SlowFlush*() function

The status table indicates the content of each hash entry—whether a hash entry is empty, contains a partial aggregate for the new window, or a potential aggregate for the previous, old window. As the *ProcessTuple* function shows, on hash-table collision, if the existing aggregate belongs to the old window, it is output and the slot is used for the new aggregate. However, a problem occurs if the existing aggregate belongs to the new window. Because low-level aggregation must preserve output order, it must first flush all the aggregates of the old window before it can output the existing colliding aggregate.[4] Therefore, because it must satisfy the order requirement, slow-flush may not effectively smooth out the output of the low-level aggregate, especially when the number of groups is large. Flushing the hash table can create a workload surge during which incoming tuples cannot be processed, limiting the maximum stream rate supported by IOP. In general, slow-flush intentionally increases

[4] The deployed version of Gigascope actually uses a better replacement policy—if the existing aggregate belongs to the new window, *ProcessTuple* also checks the next hash entry to see whether it can accommodate the new aggregate without flushing all old aggregates.

result latency to smooth out the workload, but the amount of latency that IOP can introduce for that purpose is very limited, due to its order-maintenance requirement.

In contrast to IOP with slow-flush, OOP may permit much higher throughput. The most important benefit of OOP in terms of workload smoothing is that, as it has no order requirement, the low-level aggregation does not need to flush all partial aggregates from the previous window when two aggregates from the new window collide. In more detail, suppose the desired maximum low-level latency is *m* windows. OOP can address workload smoothing in two ways. First, it may use lazy-flush, which simply relies on hash table collisions to naturally flush old aggregates, but with a check that aggregates are flushed with a maximum delay of *m* windows. Alternatively, OOP can also explicitly use slow-flush. OOP with slow-flush outputs one old aggregate every *i* new packets, and guarantees a maximum result delay of *m* windows. Both *i* and *m* are tunable parameters of the low-level sub-aggregation. As we show in our performance study, both OOP with lazy-flush and OOP with slow-flush achieve better throughput than IOP with slow-flush when there is a large number of groups.

8.3.4. Discussion

OOP is a more scalable architecture, especially in distributed computing environments, where the input data for a query operator may come from different processors, or even different machines far from one another. An issue with IOP in such an environment is that operators can be blocked due to network congestion and routing problems of a single processor. For example, a TCP connection might break

and need to be re-instantiated. These network problems can cause a significant delay and even hang an IOP system. In addition, even when the network is reliable, enforcing order on data coming from multiple processors may incur prohibitive memory and latency costs due to variations in data transmission delays and processor workloads.

OOP is also a more permissive architecture that can accommodate operator implementations that require out-of-order processing. For example, to improve throughput, stream systems may want to process tuples out of order. Avnur and Hellerstein propose an adaptive query processing mechanism, called Eddies, that dynamically routes tuples to query operators based on operator load [6]. To improve interactive query performance, Franklin et al. [50] propose algorithms that re-order tuples based on their importance. Further, the OOP architecture potentially opens new options for query optimization. In traditional database systems, one logical operator may have multiple physical implementations and the system may choose among them based on the properties of input relations. Similarly, OOP systems can potentially choose among different physical implementations of logical query operators based on properties of input streams. For example, the WID implementation generally has quite low memory requirements. However, in situations where the number of tuples per window is small, the size of partial aggregates is large, and some tuples are late enough to keep several windows open, then the memory cost of WID may exceed that of the buffered implementation. In such situations, a buffered implementation that

processes windows sequentially may be preferable, although it incurs more delay for enforcing tuple order and computing aggregates.

8.4. Experimental Evaluation

In this section, we present an experimental study of our OOP implementations in two stream systems, Gigascope and NiagaraST. We converted a version of Gigascope to OOP and term the converted systems OOP-Gigascope. We also implemented IOP query operators in NiagaraST for the purpose of these experiments and term it IOP-NiagaraST.

8.4.1. Performance Study of OOP with Gigascope

The experiments with Gigascope were conducted using network feeds generated by the RouterTester® traffic generator. RouterTester is a product of Agilent Technologies Inc. The traffic generator can generate multiple streams of IP traffic, and the content of each stream can be configured, including the number of packets per second. Our focus was to evaluate the memory and throughput benefits of OOP over high-speed streams. Each experiment is running until the measurements stabilize. All experiments were conducted on a dual-processor dual-core Intel® Xeon™ CPU 2.80GHz processor with 4 GB of RAM running Linux 2.4.21.

Experiment 1: This experiment shows how OOP can improve throughput during workload surges, and uses the following query, Q8-1, which computes the number of packets from a network interface for each (srcIP, destIP)-pair for every minute.

Q8-1: “Count the number of packets from each source and destination IP pair in the Main link for the past minute; update the results every minunte.”

```
SELECT      srcIP, destIP, count(*)
            [RANGE 1 minute, SLIDE 1 minute, WA ts]
FROM        Main
GROUP BY    scIP,
destIP
```

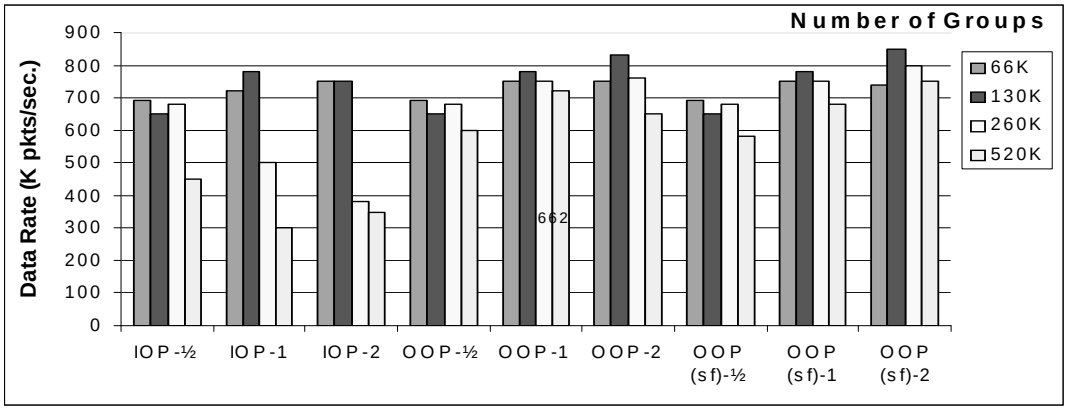


Figure 8-7 Throughput comparison of IOP and OOP for a count query, Q8-1, using Gigascope

We executed Q8-1 with Gigascope and OOP-Gigascope, varying the number of groups in the stream and the size of the hash table used by the low-level sub-aggregation. In addition, we experimented with two OOP implementations of the low-level sub-aggregation—with slow-flush and lazy-flush. We measured the maximum stream rate that Gigascope and OOP-Gigascope could support without dropping tuples, by incrementally increasing the stream rate by 5K packets per second until the query starts dropping tuples. The number of groups was varied from 66K to 520K—more groups mean fewer tuples per group but more work when outputting results at window boundaries; the low-level hash-table size was dependent on the number of groups. For each case, we used three hash table sizes: half, equal to, and twice the

number of groups. A smaller hash table means more collisions. A larger hash table means fewer collision and thus fewer groups evicted before a window completes, but more groups evicted at flush time. OOP may use either lazy-flush or slow-flush to improve workload smoothing and thereby throughput, although it may introduce a small amount of latency. In this experiment, both OOP-Gigascope with lazy-flush and slow-flush allow an extra delay of two windows to spread the workload across window boundaries. Further, OOP-Gigascope with slow-flush explicitly flushes an aggregate for an old window every 160 incoming packets. In contrast, Gigascope uses an aggressive slow-flush, explicitly flushing an aggregate once per incoming tuple.

Table 8-1 CPU Usage Comparison: OOP vs. IOP

num of grps / exp. config	66K	130K	260K	520K
IOP-1/2	99%	99%	99%	68%
OOP-1/2	99%	99%	99%	95.6%
OOP (sf)-1/2	99%	99%	99%	94.6%
IOP-1	95.5%	97%	70.5%	51%
OOP-1	98.8%	99%	99%	94%
OOP (sf)-1	99%	99%	97.8%	91%
IOP-2	96.3%	85%	55.6%	50%
OOP-2	97.2%	97.9%	95.2%	95.6%
OOP (sf)-2	96.2%	99%	97.3%	96.7%

Figure 8-7 shows the results of this experiment. Therein, OOP and OOP (sf) represent the OOP implementations of low-level sub-aggregation without slow-flush (lazy-flush) and with slow-flush, respectively. Values ½, 1, and 2 indicate the relative size

of the hash table in the sub-aggregation. In addition to measuring the maximum supported data rate, we also measured CPU utilization of the low-level query—the data rate, the number of groups, and hash table size all affect CPU utilization. (The data rate of the high-level query is much lower than that of the low-level query and thus the CPU utilization of the high-level query is always much lower than that of the low-level query.) Table 8-1 shows the peak CPU usage for each query run. As it shows, when the number of groups is large and with a sufficient number of hash table entries, the CPU utilization of the IOP approach for the maximum data rate that it can support without dropping tuples is much less than its OOP counterparts, which indicates that it can only support a much lower stream rate than those counterparts. When the number of groups is small, for example 66K, the stream rates that IOP and OOP can support are about the same, and the CPU utilizations are all close to saturation. However, when the number of groups is large, with a reasonable hash-table size, OOP can support a much higher stream rate than IOP. For example, at 260K groups, with 540k hash table entries, OOP and OOP (sf) can support 760K pkts/sec and 800k pkts/sec, respectively, while IOP can only support 400K pkts/sec. However, an overly large hash table may adversely affect the throughput of the query because it increases the workload for hash table flush, especially for the IOP cases. With 520K groups, IOP-2 only supports a maximum data rate of 350K packets per second, with CPU utilization of 50% and IOP-1 supports only 300K packets per second with CPU utilization of 51%. As we have discussed, when there is a hash-table collision between an incoming packet and an existing aggregate from the current window, IOP needs to

flush all aggregates from the previous window before any new tuples can be processed. During the hash table flush, new packets in the ring buffer are not processed and packets will be dropped if the ring buffer fills. When the number of groups is large and the data rate is high, an overly large hash table (over a million entries in our example) causes an increase in the number of aggregates to be flushed (a larger workload surge), and reduces the data rate that IOP can support without dropping tuples. The low CPU utilization for IOP with an overly large hash table is associated with the low data rates that IOP can support in these cases. An overly large hash table means a large number of groups to output and thus much more work during flush, and so the instantaneous peak CPU usage can get to 100%, while the average CPU usage far from saturated. OOP is generally better than IOP, especially for streams with large numbers of groups, and is less sensitive to the hash-table size.

Experiment 2: This experiment examines the potential memory-usage benefits of OOP for aggregation queries monitoring multiple data sources, using the following query, Q8-2, which computes the number of packets for each source and destination IP address of M1 and M2 links.

Q8-2: "Count the number of packets from each source and destination IP pair in M1 and M2 links for the past minute; update the results every minunte."

```
SELECT      srcIP, destIP, count(*)
            [RANGE 1 minute, SLIDE 1 minute, WA ts]
FROM        M1 UNION M2
GROUP BY    srcIP, destIP
```

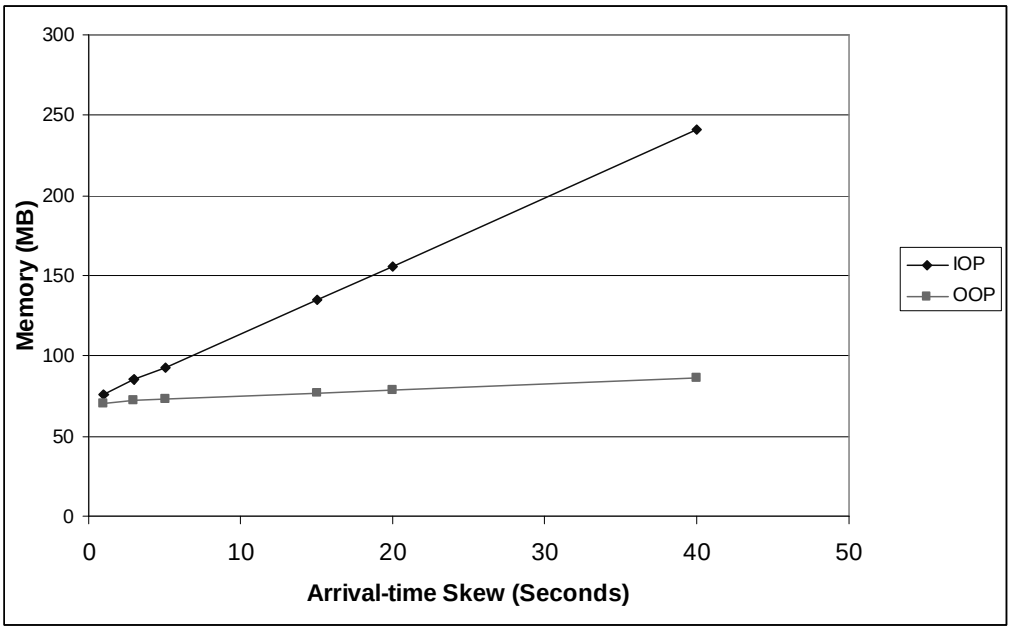


Figure 8-8 Comparison of memory usage for OOP- and IOP-Gigascope on a window-count query over the union of two streams (Q8-2), for varying skew

The rates of M1 and M2 are both 110k pkts/sec, and the total number of groups in them is 65,536. We varied the arrival-time skew of M1 and M2 from 0 to 40 seconds, and executed Q3 with both Gigascope and OOP-Gigascope, recording the maximum memory usage. (We ran each query several hours until its memory usage stabilized.) Figure 8-8 shows the results of this experiment. OOP generally uses less memory than the original IOP version of Gigascope; as arrival-time skew increases, the memory usage of OOP remains relatively flat, while that of IOP increases dramatically.

Experiment 3: This experiment provides a comparison of memory usage in Gigascope for a tumbling-window join query, Q8-3, with a window size of 10 seconds and input from multiple sources. Q8-3 joins packets within the same NetFlow (a

network connection between a pair of srcIP and srcPort, and destIP and destPort), but in opposite directions.

Q8-3: "Find the network packet pairs from the union of A with B and the union of C with D that are in correponding NetFlow for each 10-minute interval."

```
SELECT    M1.srcIP, M1.destIP, M1.ts
FROM      A UNION B as M1, C UNION D as M2
          [RANGE TUMBLING 10 seconds, WA ts],
WHERE     M1.srcIP = M2.destIP and M1.destIP and M2.srcIP and
          M1.srcPort = M2.destPort and M1.destPort = M2.srcPort
```

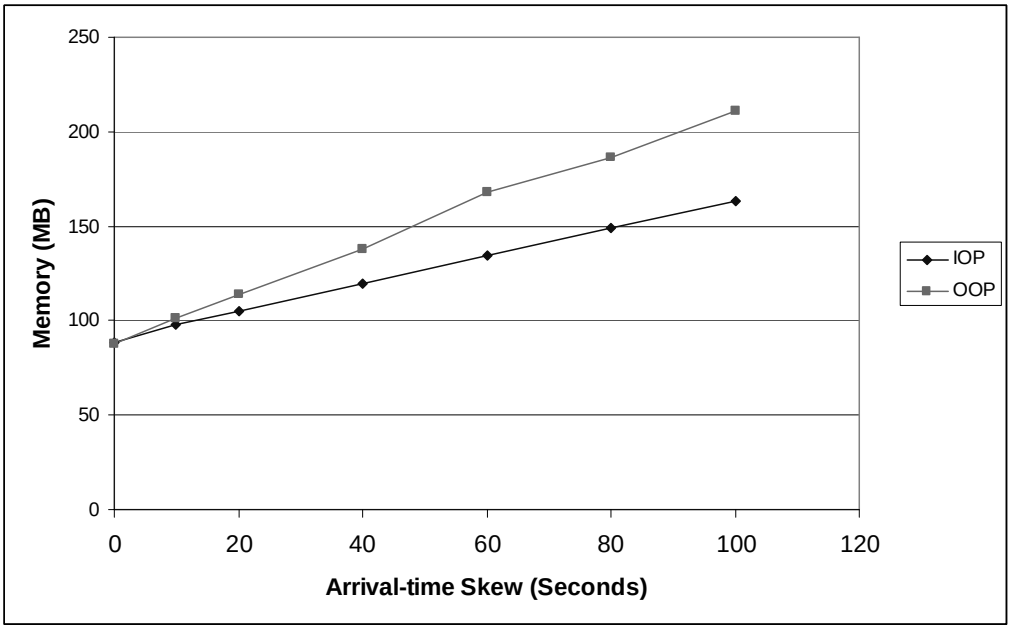


Figure 8-9 Memory comparison of IOP and OOP evaluation for a tumbling-window join query, Q8-3, with arrival skew of different input streams

Each input to the join operator is a union of two streams: Union(A, B), and Union(C, D). Q8-4 specifies a 10-second tumbling-window join condition in Gigascope. The

rate of each stream is 10K pkts/sec. (In practical stream-join queries, the input rates are often relatively low, because of prior data reduction by sampling or aggregation.) We varied the arrival-time skew of A–B, and C–D from 0 to 100 seconds, and recorded the maximum memory usage of each query run. Figure 8-9 shows the results of this experiment. The number of tuples that the IOP and OOP approaches need to maintain is the same. The difference is that in the IOP version, the tuples reside in input buffers of merge operators; in the OOP version, they are held in join hash tables. This experiment shows that while there is structural overhead for the Gigascope implementation of OOP join, compared to its IOP implementation, the overhead is not severe. When the arrival skew is below 20 seconds, the memory overhead is inconsequential. In this experiment, the OOP join uses at most 20% more memory than the IOP case. Our OOP join implementation used the hash-table structure of the original IOP join, which was not optimized for memory overhead.

8.4.2. OOP with NiagaraST

Experiments in NiagaraST were conducted on a Dual-Core AMD Opteron™ Processor 2214 with 4GB main memory, running Ubuntu Linux 2.6.17-10-server, and Sun® Java VM 1.5.

Data Generation: For our experiments, we generated stream sources of different data volumes and different time skews using network packet headers from the Passive Measurement and Analysis project [48]. We generated three streams, two with high volume (approximately 4000 tuples/second), called M1 and M2, and one with very

low volume (less than a tuple per second), called C. The total data set size is approximately 135 MB. We simulated time skews among M1, M2, and C by manipulating the placement of tuples in the data file used to generate the three streams.

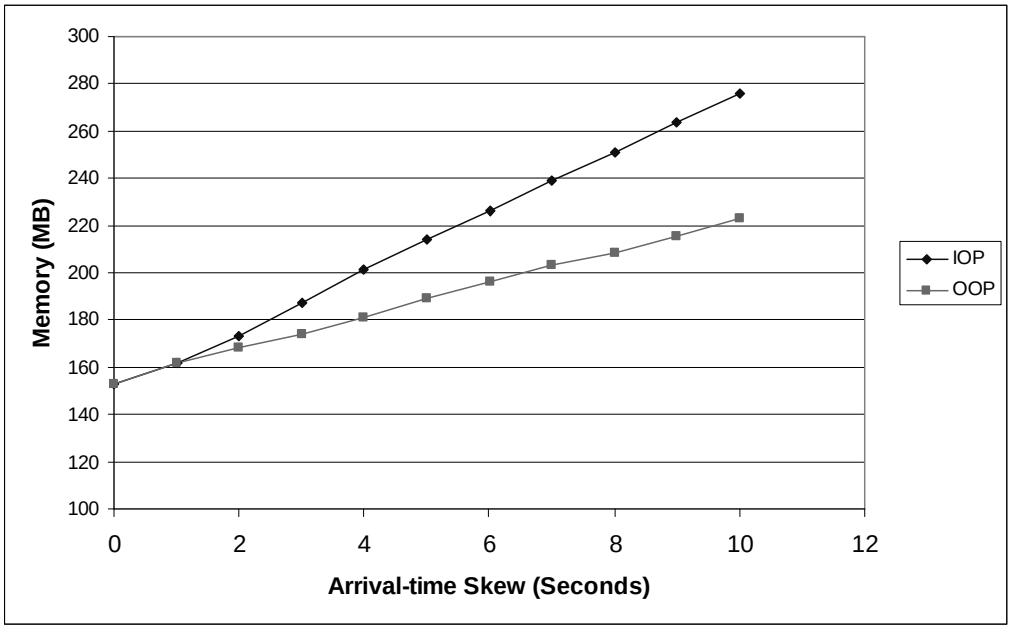


Figure 8-10 Memory comparison of IOP and OOP evaluation in NiagaraST for a tumbling-window join query, Q8-4, with late tuples on one input

Experiment 1: This experiment compares memory usage of IOP and OOP for an equality-join query on progressing attributes, Q8-4, in IOP-NiagaraST and NiagaraST. Q8-4 is a tumbling-window join that joins packets within the same NetFlow but in opposite directions. One input of the join contains late tuples, which is simulated by combining M2 with a version of C that is skewed late. The delay of C varies from 0

second to 10 seconds. Figure 8-10 shows that with the increase in the delay of the late tuples from 0 to 10 seconds, the memory use for OOP increases more slowly than for IOP. (Although Figure 8-10 looks similar to Figure 8-9, the range of the a-axis values of the two figures are significantly different—the x-axis of Figure 8-9 ranges from 0 to 120 seconds, which is much broad than Figure 8-10.) The memory advantage of OOP comes from the Join operator in OOP purging M2 tuples sooner than in IOP.

Q8-4: "Find the network packet pairs from the M1, and the union of M2 and C, that are in corresponding NetFlows for each 1 minute interval."

```
SELECT    M1.srcIP, M1.destIP, M1.ts
FROM      M1, M2 Union C as M3
          [RANGE TUMBLING 1 minute, WA ts]
WHERE     M1.srcIP = M3.destIP and M1.destIP = M3.srcIP and
          M1.srcPort = M3.destPort and M1.destPort = M3.srcPort
```

Experiment 2: This experiment compares memory performance of IOP and OOP on a sliding-window aggregate query over multiple sources, Q8-5, in IOP-NiagaraST and NiagaraST. Q8-5 computes the sliding-window count of packets over a UNION of M1, M2 and C.

Q8-5: "Count the number of packets in M1, M2 and C links for the past 5 minutes; update the results every minunte."

```
SELECT    count(*)
          [RANGE 5 minutes, SLIDE 1 minute, WA ts]
FROM      M1 UNION M2 UNION C
```

We varied the delay of the arrival of C from 0 to 10 seconds, and measured the maximum memory usage. Figure 8-11 shows that the memory usage of IOP grows significantly as the delay of C increases, while that of OOP is relatively stable. Here,

the memory benefit of OOP is due to OOP aggregation directly reducing tuples into aggregates without first buffering and sorting the input.

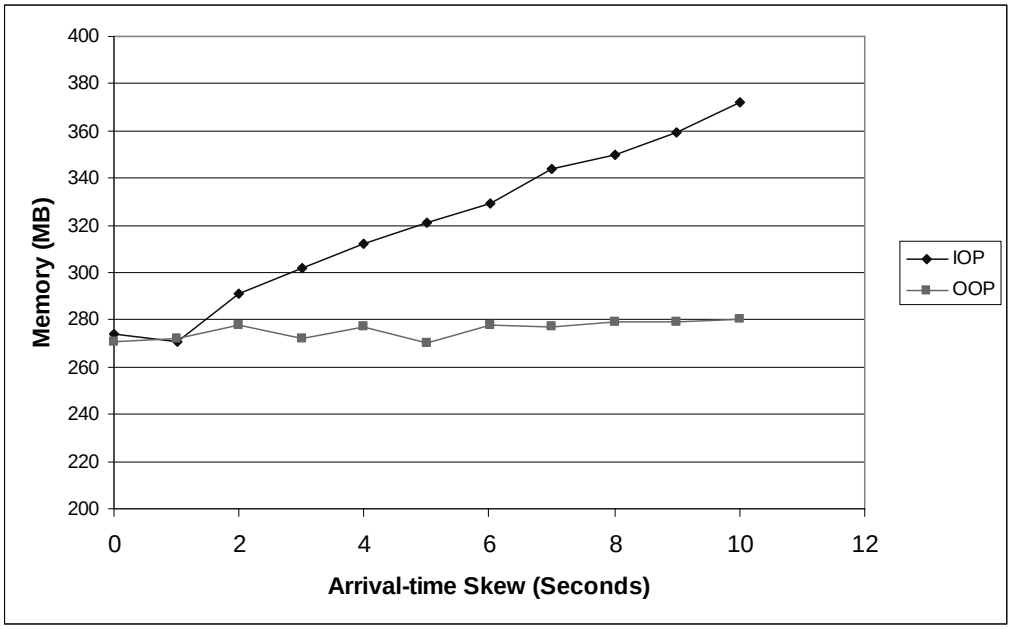


Figure 8-11 Memory comparison of IOP and OOP in NiagaraST for a sliding-window count, Q8-5, with arrival-time skew among multiple data

Discussion: Our experience with OOP architectures is encouraging. We have seen improvement over IOP in memory, latency and throughput under a variety of conditions. The fact that improvements were seen in two substantially different stream systems, NiagaraST and Gigascope, suggests that the benefits of OOP are widely applicable. The implementation overhead for supporting OOP does not seem severe, recognizing that any practical stream system will need a stream-progress mechanism beyond just tuple arrival, so that lulls in one input stream do not completely stall the stream system.

Chapter 9

CONCLUSION AND FUTURE WORK

This thesis focuses on more flexible and more efficient evaluation of window stream queries. We observed that the evaluluation of window stream queries can utilize information on stream progress, and does not require ordered streams. Based on this observation, our work in this thesis removes the order requirement for stream systems by introducing order-insensitive implementations of windowed query operators and a new architecture for stream systems.

In this thesis, we start with a new data model for streams, the progressing-stream model. Instead of requiring ordered streams, the progressing-stream model separates stream progress from physical-arrival order and only requires that streams have a progressing attribute. Then, assuming progressing streams, we present window-semantics definitions for a window aggregation and window join. In our definition, window semantics are defined on the window specification and the progressing attribute value of tuples in the streams and need not rely on any physical stream arrival properties. The window-semantics definitions lay the foundation for the order-insensitive implementations of window aggregation and window join. We present three implementation algorithms for window aggregation: the WID implemenetation, which is directly based on our window semantics definition, the Paned-WID implementation, which optimizes the execution time for sliding-window aggregation by sharing sub-aggregates, and AdaptWID, which optimizes the memory usage for

input with data distribution skew. We also present order-insensitive evaluation algorithms for tumbling-window join and sliding-window join. These order-insensitive implementations leverage punctuation to indicate ends of window extents. They not only naturally accommodate out-of-order input, but also perform better than their order-sensitive counterparts, especially in terms of memory usage and latency.

The order-insensitive implementations of stream query operators allow us to move to a new architecture for stream systems, OOP (*O*ut-of-*O*rder *P*rocessing). OOP is in contrast to IOP (*I*n-*O*rder *P*rocessing), which is the existing architecture that many stream systems assume. The key idea of the OOP architecture is explicitly communicating stream progress to query operators and thus freeing query operators from the burden of order maintainence. We use punctuation as the mechanism to explicitly communicate stream progress in our implementation of OOP—propagating punctuation is part of query operator implementations. We experimented with the OOP architecture in two stream systems, Gigascope and NiagaraST, and performance results from both systems are encouraging.

Here we also briefly discuss the tradeoffs of the OOP architecture. Having explicit information on stream progress indicates overheads in OOP systems, in both system implementation and query execution. In the implementation of OOP systems, query operator implementations need to support punctuation processing. For example, with OOP, the implementation of window aggregation must support outputting results and purging state based on punctuation, while with IOP, as window aggregation processes a window extent at a time, outputting results and purging state can be easily

implemented by flushing the hash table. However, making stream progress explicit also simplifies some operator implementations, as maintaining stream order is not needed in OOP systems. For example, without maintaining output order, the implementation of the (bag) Union operator with the OOP is much simpler than IOP. In query execution, punctuation may increase the volume of streams and thus increase query processing time and consume transmission bandwidth in distributed stream systems. However, Tucker observed very limited punctuation-processing overhead even with punctuation-to-tuple ratios as high as 15%. These results assume that punctuation is grammatical. Otherwise, query operators (or, at least the input operators) also need to block any tuples violating punctuation, which induces increased computational cost per tuple.

The OOP architecture allows a wider range of options for stream query evaluation, and thus can lead to other interesting topics. First, our current implementations always produces accurate results. However, if the amount of disorder is large, the latency that it takes to produce accurate results may not be tolerable for real-time applications. It is interesting to consider extending the current OOP architecture to support speculative results that are approximate, but can be produced earlier than accurate results, and then revisions that correct the speculative results. Second, stream-query optimization is also interesting in the OOP architecture in that the effects of disorder must be considered in cost models for comparing alternative query plans. Third, we have proposed an adaptive algorithm for aggregation to deal with varying data distributions. Adaptive algorithms for window join to deal with varying stream properties such as data

distribution skew and arrival-time delays are also desirable. Overall, we believe that our work allows stream query evaluation to be more flexible and potentially opens up other research topics.

REFERENCES

[1] The Abilene Observatory. http://abilene.internet2.edu/observatory. 2005.

[2] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, X., and Zdonik, S. The design of the Borealis stream processing engine. In *Proceedings of 2nd Biennial Conference on Innovative Data Systems Research* (CIDR 2005), Asilomar, CA, 2005.

[3] Arasu, A., Babu, S. and Widom, J. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 14, 1, 2005.

[4] Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12, 2, 2003.

[5] Arasu, A., and Widom, J. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the 2004 International Conference on Very Large Databases* (VLDB 2004), Toronto, Canada, 2004.

[6] Avnur, R., and Hellerstein, J. M. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (SIGMOD 2000), Dallas, Texas, 2000.

[7] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. Models and issues in data stream systems. In *Proceeding of the 2002 ACM Symposium on Principles of Database Systems* (PODS 2002), Madison, Wisconsin, 2002.

[8] Barga, R., Goldstein, J., Ali, M., and Hong, M. Consistent streaming through time – a vision for event stream processing. In *Proceedings of 3nd Biennial Conference on Innovative Data Systems Research* (CIDR 2007), Asilomar, CA, 2007.

[9] Bolot, J. End-to-end packet delay and loss behavior in the internet. In *Proceeding of the 1993 ACM SIGCOMM International Conference*, San Francisco, CA, 1993.

[10] Carney, D., Cetintemel, U., Cheniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. Monitoring streams – A new class of data management applications. In *Proceedings of the 2002 International Conference on Very Large Databases* (VLDB 2002), Hong Kong, China, 2002.

[11] Chandrasekaran, S., and Franklin, M. J. PSoup: A system for streaming queries over streaming data. *The VLDB Journal*, 12, 2, 2003.

[12] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In

Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, CA, 2003.

[13] Cormode, G., Johnson, T., Korn, F., Muthukrishnan, S., Spatscheck, O., and Srivastava, D. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (SIGMOD 2004), Paris, France, 2004.

[14] Cranor, C., Johnson, T., and Spatashek, O. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 2003), San Diego, CA, 2003.

[15] Ding, L., and Rundensteiner, E. A. Evaluating window joins over punctuated streams. In *Proceeding of the 2004 International Conference on Information and Knowledge Management* (CIKM 2004), Washington, DC, 2004.

[16] Rundensteiner, E.A, Ding, L., Sutherland, T., Zhu, Y., Pielech, B. and Mehta, N. CAPE: Continuous query engine with heterogeneous-grained adaptivity. In *Proceedings of the 2004 International Conference on Very Large Databases* (VLDB 2004), Toronto, Canada, 2004.

[17] Duda, R. O, Hart, P. E., and Stork, D. G. *Pattern Classification. Second edition.* A Wiley-Interscience Publication. Pages 239-242, 2001.

[18] Elfeky, M. G., Aref, W.G., and Elmagarmid, A.F. Using convolution to mine obscure periodic patterns in one pass. In *Proceeding of the 2004 International Conference on Extending Database Technology* (EDBT 2004), Heraklion, Crete, Greece, 2004.

[19] Gabaix, X., Gopikrishnan, P., Plerou, V., and Stanley, H.E. A theory of power law distributions in financial market fluctuations. *Nature*, 423, 267–270, 2003.

[20] Gedik, B., Andrade, H., Wu, K., Yu, P., and Doo, M. SPADE: The system S declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 2008), Vancouver, Canada, 2008.

[21] Golab, L. and Ozsu, M. T. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 2003 International Conference on Very Large Databases* (VLDB 2003), Berlin, Germany, 2003.

[22] Golab, L., and Özsu, T.M. Update-pattern-aware modeling and processing of continuous queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 2005), Baltimore, MD, 2005.

[23] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery* 1, 1, 29-53, 1997.

[24] Greenwald, M., and Khanna, S. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 2001), Santa Barbara, CA, 2001.

[25] Hammad, M. A., Aref, W. G., and Elmagarmid, A, K. Optimizing in-order execution of continuous queries over streamed sensor data. In *Proceeding of the 17th International Conference on Scientific and Statistical Database Management* (SSDBM 2005), Santa Barbara, CA, 2005.

[26] Hammad, M., Aref, W., Franklin, M., Mokbel, M., and Elmagarmid, A.K. Efficient execution of sliding window queries over data streams. Purdue University Department of Computer Sciences Technical Report Number CSD TR 03-035, December 2003.

[27] Hammad, M., Franklin, M., Aref, W., and Elmagarmid, A. Scheduling for shared window joins over data streams. In *Proceedings of the 2003 International Conference on Very Large Databases* (VLDB 2003), Berlin, Germany, 2003.

[28] Hammad, M.A. Mokbel, M.F. Ali, M.H. Aref, W.G. Catlin, A.C. Elmagarmid, A.K. Eltabakh, M. Elfeky, M.G. Ghanem, T.M. Gwadera, R., Ilyas, I.F., Marzouk, M., and Xiong, X. Nile: A query processing engine for data streams. In *Proceeding of the 2004 International Conference on Data Engineering* (ICDE 2004), Boston, MA, 2004.

[29] Hwang, J-H, Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. High-Availability Algorithms for Distributed Stream Processing. In *Proceddings of 2005 International Conference on Data Engineering* (ICDE'05), Tokyo, Japan, 2005.

[30] Hwang, J-H., Cetintemel, U. and Zdonik, S. Fast and Highly-Available Stream Processing over Wide Area Networks, In *Proceedings of 2008 International Conference on Data Engineering* (ICDE 2008) Cancun, Mexico, 2008

[31] Ives, Z. G., Florescu, D., Friedman, M., Levy, A., and Weld, D. S. An adaptive query execution system for data integration. In *Proceedings of the 1999 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 1999), Philadelphia, PA, 1999.

[32] Jagadish, H. V., Mumick, I. S., and Silberschatz, A. View maintenance issues for the Chronicle data model. In *Proceeding of the 19995 ACM Symposium on Principles of Database Systems* (PODS 1995), San Jose, CA, 1995.

[33] Johnson, T., Muthukrishnan, S., Shkapenyuk, V., and Spatscheck, O. A heartbeat mechanism and its application in Gigascope. In *Proceedings of the 2005 International Conference on Very Large Databases* (VLDB 2005), Trondheim, Norway, 2005.

[34] Kabra, N. and DeWitt, D. J. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 1998), Seattle, Washington, 1998.

[35] Kang, J., Naughton, J. F., and Viglas, S. Evaluating window joins over unbounded streams. In *Proceedings of the 2003 International Conference on Data Engineering* (ICDE 2003), Bangalore, India, 2003.

[36] Keys, K., Moore, D., and Estan, C. A robust system for accurate real-time summaries of internet traffic. *ACM SIGMETRICS Performance Evaluation Review* 33, 1, June 2005.

[37] Krishnamurthy, S., Wu, C., and Franklin, M. On-the-fly sharing for streamed aggregation. In *Proceeding of the 2006 ACM International Conference on Management of Data* (SIGMOD 2006), Chicago, Illinois, 2006.

[38] Law, Y.-N., Wang, H., and Zaniolo, C. Query languages and data models for database sequences and data streams. In *Proceedings of the 2004 International Conference on Very Large Databases* (VLDB 2004), Toronto, Canada, 2004.

[39] Li, H.-G., Chen, S., Tatemura, J., Agrawal, D., Kandan, S., and Hsiung, W-P. Safety guarantee of continuous join queries over punctuated data streams. In *Proceedings of the 2006 International Conference on Very Large Databases* (VLDB 2006), Seoul, Korea, 2006.

[40] Li, J., Maier, D., Tufte, K., and Papadimos, V. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34, 1, 2005.

[41] Li, J., Maier, D., Tufte, K., and Papadimos V. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 2005), Baltimore, MD, 2005.

[42] Li, J., Tufte, K., Shkapenyuk, V., Papadimos, V., Johnson, T., and Maier, D. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. In *Proceedings of the 2008 International Conference on Very Large Databases* (VLDB 2008), Auckland, New Zealand, 2008.

[43] Li, W. Random texts exhibit Zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, 38, 6, 1992.

[44] Madden, S., Shah, M., Hellerstein, J. M., and Raman, V. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM International Conference on Management of Data* (June 2002), Madison, Wisconsin, 2002.

[45] Mazzucco, M., Ananthanarayan, A., Grossman, R. L., Levera, J., and Rao, G. B. Merging multiple data streams on common keys over high performance networks. In *Proceedings of the IEEE/ACM SC2002 Conference*, Baltimore, MD, 2002.

[46] Naughton, J.F., DeWitt, D.J., Maier, D., Aboulnaga, A., Chen, J., Galanis, L., Kang, J., Krishnamurthy, R., Luo, Q., Prakash, N., Ramamurthy, R., Shanmugasundaram, J., Tian, F., Tufte, K., Viglas, S., Wang, Y., Zhang, C., Jackson, B., Gupta, A., and Chen, R. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, 24, 2, 2001.

[47] Nagaraj, K., Naidu, K.V.M., and Rastogi, R. Effecient Aggregate computation over data streams. In *Proceedings of the 2008 International Conference on Data Engineering* (ICDE 2008), Cancun, Mexico, 2008.

[48] Passive Measurement and Analysis project. San Diego Supercomputer Center. http://pma.nlanr.net/PMA. 2005.

[49] Radiation Detection Center, Lawrence Livermore National Lab. http://rdc.llnl.gov. 2005.

[50] Raman, V., Raman, B., and Hellerstein, J. M. Online dynamic reordering for interactive data processing. In *Proceedings of the 1999 International Conference on Very Large Databases* (VLDB 1999), Edinburgh, Scotland, UK, 1999.

[51] Sharaf, M.A, Chrysanthis, P.K., Labrinidis, A., and Pruhs, K. Efficient Scheduling of Heterogeneous Continuous Queries. In *Proceeding of the 2006 Very Large Databases Conference* (VLDB 2006), Seoul, Korea, 2006.

[52] Sharaf, M.A., Chrysanthis, P.K., and Labrinidis, A. Preemptive Rate-based Operator Scheduling in a Data Stream Management System, In *the Proceedings of the Third ACS/IEEE International Conference on Computer Systems and Applications* (AICCSA'05), Cairo, Egypt, 2005.

[53] Sharaf, M.A., Labrinidis, A., Chrysanthis, P.K., and Pruhs, K. Freshness-Aware Scheduling of Continuous Queries in the Dynamic Web In *Proceeding of the 2005 International ACM Workshop on the Web and Databases* (WebDB 2005), Baltimore, Maryland, 2005.

[54] Schreier, U., Pirahesh, U., Agrawal, R., and Mohan, C. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proceedings of the 1991 International Conference on Very Large Data Bases* (VLDB 1991), Barcelona, Catalonia, Spain, 1991.

[55] Segev, A., and Shoshani, A. Logical modeling of temporal data. In *Proceedings of the1987 ACM SIGMOD Annual Conference* (SIGMOD 1987), San Francisco, CA, 1987.

[56] Seshadri, P., Livny, M., and Ramakrishnan, R. Sequence query processing. In *Proceedings of the 1994 International Conference on Management of Data* (SIGMOD 1994), Minneapolis, MN, 1994.

[57] Seshadri, P., Livny, M., and Ramakrishnan, R. SEQ: A model for sequence databases. In *Proceedings of the 1995 International Conference on Data Engineering* (ICDE 1995), Taipei, Taiwan, 1995.

[58] Seshadri, P., Livny, M., and Ramakrishnan, R. The design and implementation of a sequence database system. In *Proceedings of the 1996 International Conference on Very Large Databases* (VLDB 1996), Mumbai, India, 1996.

[59] Shah, M., Madden, S., Franklin, M., and Hellerstein, J. Java support for data-intensive systems: Experiences building the telegraph dataflow system. In *SIGMOD Record*, 30, 4, 2001.

[60] Shapiro, L. D. Join processing in database systems with large main memories. *ACM Transaction on Database Systems*, 11, 3, 1986.

[61] Srivastava, U., and Widom, J. Flexible time management in data stream systems. In *Proceeding of the 2004 ACM Symposium on Principles of Database Systems* (PODS 2004), Paris, France., 2004

[62] Srivastava, U, and Widom, J. Memory-Limited Execution of Windowed Stream Joins. In *Proceedings of the 2004 International Conference on Very Large Databases* (VLDB 2004), Toronto, Canada, 2004.

[63] The STREAM Group. STREAM: The Stanford STREAM Data Manager. *IEEE Data Engineering Bulletin*, 26, 1, 2003.

[64] Stanford Stream Query Repository. http://www-db.stanford.edu/stream/sqr. 2005

[65] StreamBase. http://www.streambase.com/. 2008.

[66] Stonebraker, M., Cetintemel, U., and Zdonik, S. The 8 requirements of real-time stream processing. In *Proceeds of 2005 International Conference on Data Engineering* (ICDE 2005), Tokyo, Japan, 2005.

[67] Sullivan, M., and Heybey, A. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annul Technical Conference*, Boston, Massachusetts, 1998.

[68] Tatbul, N., Cetintemel, U., and Zdonik, S. Staying FIT: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 2007 International Conference on Very Large Databases* (VLDB 2007), Vienna, Austria, 2007.

[69] Terry, D., Goldberg, D., Nichols, D., and Oki, B. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (SIGMOD 1992), San Diego, California, 1992.

[70] Truviso. http://www.truviso.com/. 2008.

[71] Tucker, P. *Punctuated Data Streams.* Doctoral Dissertation. Oregon Health & Science University, Portland, Oregon. 2005.

[72] Tucker, P. A., Maier, D., Sheard, M., and Fegaras, L. Exploiting punctuation semantics in continuous data streams. *Transactions on Knowledge and Data Engineering*, 15, 3, 2003.

[73] Urhan, T., and Franklin, M. J. XJoin: A reactively-scheduled pipelined join sperator. *IEEE Data Engineering Bulletin*, 23, 2, 2000.

[74] Urhan, T., and Franklin, M. J. Dynamic pipeline scheduling for improving interactive query performance. In *Proceedings of the 2001 International Conference on Very Large Data Bases* (VLDB 2001), Roma, Italy, 2001.

[75] Urhan, T., Franklin, M. J., and Amsaleg, L. Cost based query scrambling for initial delays. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (SIGMOD 1998), Seattle, Washington, 1998.

[76] Viglas, S., Naughton, J. F., and Burger, J. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 2003 International Conference on Very Large Data Bases* (VLDB 2003), Berlin, Germany 2003.

[77] Wang, S., Rundensteiner, E. A., Ganguly, S., and Bhatnagar, S. State-Slice: New paradigm of multi-query optimization of window-based stream queries. In *Proceedings of the 2006 International Conference on Very Large Data Bases* (VLDB 2006), Seoul, Korea, 2006.

[78] Willinger, W., Alderson, D., and Li, L. A pragmatic approach to dealing with high-variability in network measurements. In *Proceedings of the 2004 Internet Measurement Conference* (IMC 2004), Taormina, Sicily, Italy, 2004.

[79] XMark Benchmark. http://www.xml-benchmark.org. 2005.

[80] Xing, Y., Zdonik, S., and Hwang, J-H. Dynamic load distribution in the Borealis stream processor. In *Proceeding of the 2005 International Conference on Data Engineering* (ICDE 2005), Tokyo, Japan, 2005.

[81] Zdonik, S., Stonebraker, M., Cherniack, M., Cetintemel, U., Balazinska, M, and Balakrishnan, H. The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin*, 26, 1, 2003.

[82] Zhang, R., Koudas, N., Ooi, B.C., and Srivastava, D. Multiple aggregations over data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on the Management of Data* (SIGMOD 2005), Baltimore, Maryland, 2005.