State Spill Policies for State Intensive Continuous Query Plan Evaluation

by

Mariana G. Jbantova

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

April 2007

APPROVED:

Professor Elke Rundensteiner, Thesis Advisor

Professor David Finkel, Thesis Reader

Professor Michael Gennert, Head of Department

Abstract

The needs of new modern day applications such as network monitoring systems, telecommunications data management, web applications, remote medical monitoring applications and others for near real time results over continuous data streams have spurred the development of new data management systems called Data Stream Management Systems (DSMS). Unlike traditional database systems which answer one-time user queries only after the finite data has been captured on disk, DSMSs provide on-the-fly answers to user queries as data is arriving at various rates in the form of continuous, potentially infinite streams of tuples. To meet the timeliness requirements of applications, DSMSs aim to keep all data in main memory. Thus queries with multiple stateful operators pose a major strain on memory.

Existing adaptation techniques designed to address this issue are ineffective when faced with continuous bursts of high data rates. When system load exceeds system capacity, a DSMS has three options: 1) discard some new data; 2) crash; or 3) spill data to disk. Only option three allows it to produce delayed, yet accurate and complete query results. However, this option involves disk access overhead and change in the natural order of tuples flowing through the query plan tree. As not all stream operators can process correctly out of order tuples, data spilling may have a negative impact on the quality of the final results. Moreover, since operators in a query plan are interconnected, changes in the order of tuple flows inevitably impact the stages of execution of affected downstream operators such as for example data purging . Data purging is necessary for processing continuous queries composed of stateful operators. The state of such operators is divided into finite non-overlapping sets of tuples called windows. Thus, after all the tuples for a

window have been processed and all results output, these tuples can be discarded to free memory for new data.

To address these issues, we have redesigned the state structure of continuous operators into smaller, finite, non-overlapping sets of tuples such as partitioned window groups, which incur less disk-access overhead. Second, we provide for the capability of continuous operators to correctly process out of order tuples using punctuation pointers. Third, we design methods for downstream operators to synchronize their processing stages with those of upstream operators to achieve optimized query plan throughput. Putting these techniques together, we have designed a consolidated spilling adaptation strategy which considers all aspects of operators' inter-connections in a query plan for making optimal adaptation decisions.

The effectiveness of our integrated approach was empirically tested in a comparative evaluation study against several alternate spilling adaptation strategies. We conducted our experiments on CAPE, a DSMS developed at WPI, using different types of query plans composed of multiple partitioned window join operators. Our experiments prove that despite the higher overhead of a more synchronized adaptation approach, our consolidated strategy provides better query plan performance and higher plan throughput during periods of continuous bursts of high data rates.

Acknowledgements

I would like to express my great gratitude to my advisor, Prof. Elke A. Rundensteiner for her guidance, patience, encouragement, and support. Her knowledge as a researcher and professor and her wisdom as a human being helped me to go through some very critical times of my life while working on my degree and to finish my project.

I would like to thank my thesis advisor Prof. Finkel for his time and support. I would like to express my special gratitude to Prof. Selkow for his incredible support, thoughtfulness and understanding at the times when I needed them most, and to Prof. Mike Gennert for his understanding and support.

I would like also to thank other faculty and staff members from the CS department at WPI, Prof. Mike Claypool, Glynis Hamel, Sharon Demaine, Jessica Pollock, Michael Voorhis, and Jesse Banning.

I would like to thank members of the CAPE and DCAPE team, in particular Bin Liu, Tim Sutherland, Yali Zhu, Luping Ding and Brad Momberger for their joint work on the CAPE system and for their responsiveness and help. I would like to thank the whole DSRG group and especially my officemates and friends Maged El-Sayed, Larisa Orlova, Rimma Nehme, Bin Liu, Venkatesh Raghavan, Mingzhu Wei and Ming Li for making the office a more fun place to be during the long hours of work and for their invaluable pieces of advice and encouragement at times when my spirits were low.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The rapid advances of technology accompanied with changes in market forces stimulates the development of new business and science-related applications and systems. Increase in the volumes of data available to companies and research institutions, the constantly increasing speed of networks, the continuously increasing deployment of sensors and wireless technologies in certain industries [35] and the need to analyze streams of data in real time have spurred the development of new types of data management systems called Data Stream Management Systems (DSMS). Examples of applications in need of the services offered by the newly developed systems include financial applications, network monitoring, telecommunications data management, web applications, manufacturing, remote medical monitoring applications, sensor networks, and others [35, 26, 6].

Unlike traditional database systems which deal with a finite amount of data, and answer user queries only on data that had first been captured on disk, the newly developed systems aim to answer user queries in real time as the data is arriving at various rates in the form of continuous, potentially infinite, streams of tuples. The unpredictable char-

acteristics of the arrival patterns of the data streams and the continuous nature of the queries submitted to stream processing engines, together with the constraints imposed on the streaming environment by the requirements of the applications for real-time yet accurate results necessitate the development of novel query optimization techniques. Methods developed for traditional database systems may no longer be applicable due to the specific characteristics of the streaming environment. Currently, optimization methods for DSMS include at the operator level exploiting an operator's selectivity [3, 29] using operator's punctuations for state purging [5, 29]; at the scheduler level various operators' scheduling techniques [29, 22, 4], and query approximation methods such as load shedding [35]; at the query plan level the distribution of the query plans across multiple machines [29], dynamic query plan migration [36] and operators reallocation [9, 27]. Efficiently handling critical resources such as main memory is a major concern in the design of DSMSs. Any system of computing devices whether a centralized one, consisting of a single query processor or a distributed one, consisting of multiple interconnected processors, suffers from the inevitable problem of limited resources. They all have an upper bound on the amount of data they can ultimately process at a time. Thus integration of optimization techniques at various levels of query plan processing becomes necessary for the optimal utilization of the available yet limited resources.

The focus of this thesis is the development of strategies for temporarily pushing operators' states to disk during periods of high-data arrival rates to prevent run-time memory overflow problems and potential system crashes; and strategies for bringing data back to main memory during periods of low system load or for finishing the query plan processing. Thus a DSMS's most critical resources such as main memory and processor's computing cycles can be managed and utilized in a very efficient way which would ultimately result in a better quality of service for the final applications. Such optimization techniques, however, face certain challenges. 1) The swapping of operator's states be-

tween disk and main memory has to be performed on demand at run time with little overhead. 2) The statistics on which different optimization techniques will be based has to be collected at run-time, thus it has to be light-weight, yet accurate enough for making better optimization decisions. 3) As data will be swapped between disk and main memory at run time, the system has to be able to handle out of order tuple arrivals and still produce accurate and correct results. We define as such output results with *no data tuples missed* and *no extra tuples generated*.

Adaptation techniques based on spilling data to disk and the associated with this memory management issues such as how mush data to spill and how to organize the data spilled to disk have been investigated in the design of join operators such as XJoin [32], progressive merge join and hash-merge join [23] optimized for accessing data over distributed networks stored in traditional database systems. [23] discusses the advantages and disadvantages of several different flushing policies such as flush-all policy, flush largest partition first,and flush smallest partition first policy.

Since even distributed data stream management systems have ultimately a finite amount of processing resources, data spilling can help distributed systems too ro alleviate memory shortage problems incurred by spikes in data arrival rates. [20] investigates the issues related to data spilling in a distributed environment. [20] focuses on the integration of two run-time adaptation techniques, namely, state spill to disk and state relocation to an alternate machine. The paper analyzes the tradeoffs regarding key factors affecting these two runtime operator state adaptation techniques and proposes two adaptation strategies: lazy-disk and active-disk. These strategies integrate both state spill and state relocation adaptations with different emphasis on local versus global decision making.

Unlike [20], we focus only on the issues associated with data spilling. We expand on the memory management ideas presented in [23, 32]. The techniques proposed in this work, however, are applicable to both centralized and distributed query processing

environments. The goal of the thesis is to investigate the impact of such strategies on the performance of query plans consisting of multiple state intensive operators.

1.2 Motivation

As discussed in [10], efficient processing of queries under varying data arrival rates and availabilities of system resources is the key to the success of many applications using the services of DSMS. However, current research of continuous query processing often assumes query operators with fairly-small sized operator states, for example, small window joins or stateless operators such as select and project [2, 8, 5, 16]. Despite the fact that complex multi-join continuous queries are rather common in the data integration and the data warehousing environments and the fact that there are some papers on parallel operators like FLUX [27], memory management for continuous queries with such potentially huge operator states [9] have not been carefully studied.

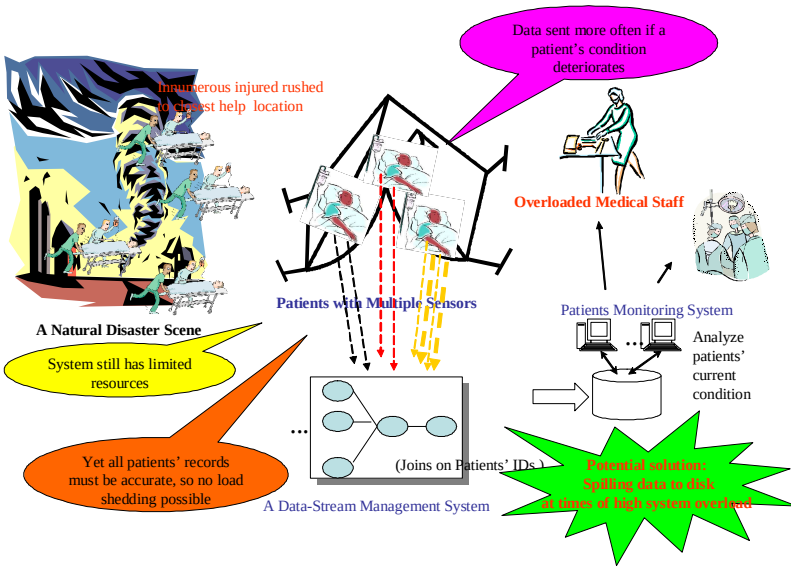

Figure 1.1: A real time mobile hospital DSMS.

For example, a data integration system may be used by medical teams working in a mobile hospital deployed to a natural disaster scene or a battlefield as illustrated on Figure 1.1. Multiple sensors per person to monitor vital life symptoms can be attached to casualties as they are checked into the hospital. Additionally, remote static database systems containing information on medical conditions or patients' health records, assuming ability to identify patients, can also be constantly queried. Assume further the ability of sensors to increase their sampling rates upon the deterioration of a patient's condition. In such situations of emergency characterized by unpredictability and chaos, the necessity of fast yet accurate decisions on behalf of the medical stuff makes it very important that the data stream management system does not become the bottleneck in the chain of events. Otherwise, this may cause the preventable death of people. Thus such a system needs to be able to operate under potentially very heavy workloads and still produce as many and accurate results as fast as possible. The more results are produced at run time the more information the medical stuff will have for making critical life-saving decisions on patients' course of treatment. Furthermore, no approximate results that we may be able to generate by the employment of load shedding techniques may be acceptable to the end-application as patients' records ought to be accurate for a later post-disaster analysis and inspection. In the context of such stringent constraints and requirements it is very important to provide an optimal main-memory management strategy.

A viable solution to the above scenario may be the temporary pushing of operators' states to disk as discussed by XJoin [32, 9] and Hash-Merge Join [23, 9]. Thus no tuples will be lost during the arrival of high bursts of data, yet the DSMS will still be able to continue to produce near real-time results regarding the current states of patients. The employment of a content-based data-spill strategy can further improve the usefulness of such a system by assigning high priority to the data of patients in critical condition and first spilling the lower-priority data: the data of patients who are currently in stable condition.

1.3 Challenges

The main challenges in the design of a join operator for the streaming context with the capacity of swapping data between disk and main memory on demand and the design of policies which would keep the operator's performance at optimal levels for a particular system load include:

1. Swapping of data between disk and main memory should not affect the accuracy of the results produced by the operator. This means there should be no missed output tuples and there should be no duplicate tuples ever output.

2. Reading and writing data to disk are very expensive operations in terms of system resource consumption. Thus, such adaptations should not be performed too often by the operator, otherwise it may hurt system performance. It is important that the moments when these adaptations need to be performed be correctly identified.

3. Since disk access is a very expensive operation, the granularity at which spilled tuples are stored on disk is important. Larger files provide for less disk access overhead, but reduce the flexibility of unspilling policies which may cause sub-optimal query plan performance. On the other hand, small files provide for more flexible unspilling policies but may cause increased disk access costs by incurring too many reads and writes.

4. To correctly detect when a DSMS experiences heavy load or when data arrival rates have slowed down and the system has enough free resources to process any spilled on disk tuples, statistics have to be collected and analyzed. Collecting statistics may prove to be counterproductive, as it is an expensive operation. It is important to decide what statistical data is sufficient and how often to collect it.

5. The partition level statistical data collected by operators should be stored in a suitable data structure.

6. Operators in a query plan are interconnected and dependent on each other. It is important that during a query plan execution operators synchronize their work to improve system performance. This entails the need for: a) propagating metadata about the stages of query execution down the query plan tree; b) policies for work synchronization; and c) policies for data purging. It should be noted that sending too much information down the query plan can hurt the performance of the operator in two ways: it will increase the time an operator spends to process the incoming information instead of processing incoming data and it will increase the amount of memory consumed by the operator.

1.4 Contributions

This thesis has made the following contributions:

1. A new partitioned window join operator with the ability to spill and unspill data to disk on demand has been designed.

2. We further extend the semantics of punctuations embedded in the data stream to encode information of the processing stages completed by an operator. Such information is used for the correct processing of out of order tuples and for the design of efficient data invalidation policies.

3. A new adaptation policy to synchronize the work of operators in a query plan has been designed. The policy uses metadata about the stages of query execution propagated down the query plan tree by operators and partition level statistical data to make better memory management adaptation decisions.

4. We have designed several different adaptation policies with different levels of query plan synchronization.

5. All the policies have been implemented and integrated into a data stream management system called CAPE.

6. Experiments on the relative performance of the different adaptation policies have been carried out using a real software system, not simulation.

Chapter 2

Preliminaries

2.1 Stateful Operators in Streaming Context

One of the distinguishing characteristics of continuous query processing is that the size of the input data may be potentially infinite. Thus query plans composed of one or more operators, which require to see the whole input before producing any query results, would not run in the streaming environment. Such operators are called blocking operators. A way of enabling queries with blocking operators to run in the streaming environment is by defining a mechanism for continuously breaking down the input stream into finite sub-streams of data. This can be achieved by imposing constraints on the query output. In the streaming context, bounds on the size of the input streams imposed by constraints on the query output are called *windows*. Windows can be defined as limits on the maximum distance tuples can be apart from each other in time or tuple count to be considered in the query. Windows are characterized by a *size* and a *sliding step*. In the rest of the thesis we will denote the size of a window with *w.size*, and its sliding step with *w.step*. A window's size can be expressed in terms of time units or tuple-counts. Consequently, there are two main types of windows: *time-based windows* and *count-based windows*. The sliding step

of a window determines the distance between two consecutive windows. Sliding steps are also called *panes* [17]. In [17] a window is said to be composed of panes.

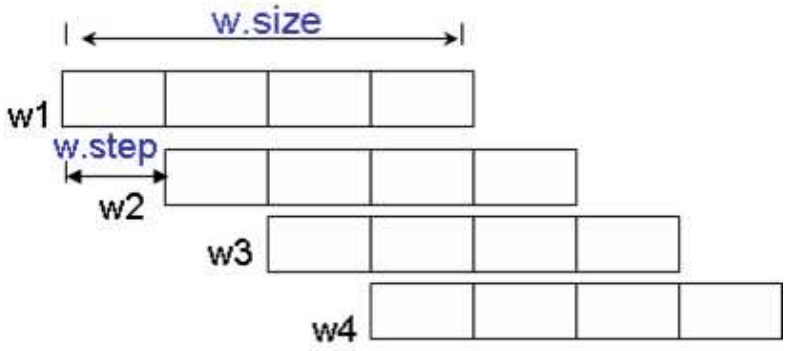

Figure 2.1: Example of a continuous window.

Based on the ratio of a window's size to its sliding step, windows can be classified as hopping or continuous. *Hopping windows* have $w.size <= w.step$ whereas continuous windows have $w.size > w.step$. In the latter case a tuple can belong to more than one window. Figure 2.1 shows an example of a sliding window with $w.size = 4$ time units and $w.step = 1$ time unit. Figure 2.2 shows an example of a hopping window with $w.size = 4$ time units and $w.step = 5$ time units . In both figures we use the notation *wid* where $id >= 1$ to identify a particular window. As it can be seen in Figure 2.2, in the case of hopping windows a tuple belongs to at most one window. [18] gives an extensive list of window types that can be imposed on an operator.

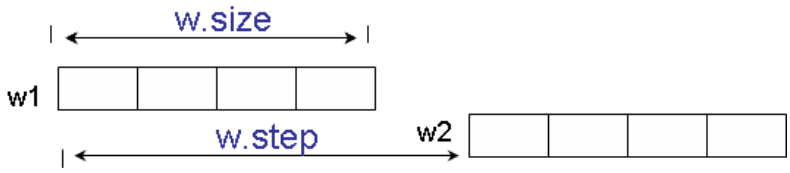

Figure 2.2: Example of a hopping window

During query processing each individual window is characterized by its beginning

and ending parameters. For time-based windows, these parameters are expressed in time units and may be calculated relatively to the beginning time of the first window. There are different ways of determining the beginning time of the first window. It can be, for instance, either set to the System time at the moment the query has been submitted for execution, or it can be set to be equal to the timestamp of the first tuple received by the operator. In this work we use the latter approach. We refer to the beginning time of the first window as the *query plan start time* and we denote it with *qp.start*. As the query plan is being executed, the newly created windows are assigned beginning, denoted with *w.start*, and ending times, denoted with *w.end*, relatively to the query plan start time. For example, let's assume that window 1 (*w1*) from Figure 2.1 has a beginning time equal to 12:00 and ending time equal to 12:04. We assume that *w.size* and *w.step* are expressed in minutes. In this example, $w.size = 4$ *minutes* and $w.step = 1$ *minute*. So window 2 will start at 12:01, one sliding step later than window 1, and it will finish at 12:05, one sliding step later than window 1. Respectively, window 3 will start at 12:02, two sliding steps later than window 1, and so on. When the state of an operator is organized in hopping windows, every new tuple that arrives at the input queues of the operator for processing will belong to at most one window at a time. On the other hand, when the state of an operator is organized in continuous windows, every new tuple that arrives for processing will belong to at least one and possibly several windows simultaneously as consecutive continuous windows have overlapping boundaries. Let us look at Figure 2.1. We again assume that the first window starts at 12:00 and ends at 12:04. Then a tuple with a timestamp of 12:03 will belong to windows 1, 2, 3 as these windows have overlapping boundaries as it is shown in Figure 2.3. The maximum number of windows (*maxW*) a tuple can belong to is determined by the following formula:

Formula 1: $maxW = \lceil (w.size/w.step) \rceil$

The windows a tuple belongs to are determined by the relative order of a tuple's ar-

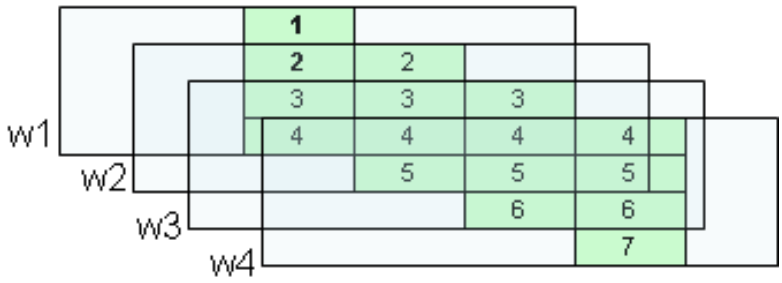

Figure 2.3: Example of maximum number of windows a tuple may belong to.

rival at the operator for processing in the case of count-based windows and by a tuple's timestamp in the case of time-based windows. As in [36], we assign a unique identification number to each window. The identification numbers are assumed to be taken from a sequence of consecutive integers starting at one. Thus a window's identification number also indicates how many sliding steps have expired from the query plan start time until the beginning time of this window, that is the number of windows preceding this one. For example, the very first window open at an operator during a query plan execution is assigned an identification number of 1. The next window whose beginning time is a sliding step away from the beginning time of the first window is assigned an identification number of 2 and so forth. We use this information to calculate the identification numbers of all windows that a newly arriving tuple would belong to. Algorithm 1 outlines the basic steps we use to do this. As Algorithm 1 shows, we first identify the ID of the last window a tuple belongs to. Then we use this information to trace back all the earlier windows the tuple belongs to. A tuple *tuple* belongs to a window *wid* if the tuple's timestamp (*tuple.time*) is within the window's boundaries, which we define here as:

Formula 2: $w.start < tuple.time <= w.end$

Windows impose constraints on the evaluation semantics of the query. Only tuples which all belong to the same window will be processed by an operator to produce valid query plan output. For example, a join operator will join a newly arriving tuple only with

Algorithm 1 Calculating the Windows a Tuple Belongs to.

```
    Input: tuple.time
    Output: set wids={n|n>=1 or emptyset}
1:  long tuple.maxW=-1
2:  long ceiling=⌈(w.time − qp.start)/w.step⌉
3:  long floor=⌊(tuple.time − qp.start)/w.step⌋
4:  long maxW.start=qp.start+ceiling*w.step
5:  long maxW.end=maxW.start+w.size
6:  long minW.end=qp.start+floor*w.step+w.size
    Case 1: tuple belongs to no windows
7:  if((w.time <= qp.start) or ((w.size < w.step and tuple.time > minW.end and maxW.start >= tuple.time) or
    (w.size < w.step and tuple.time = minW.start or tuple.time = minW.start)) then
8:      return wids = ∅
    Case 2: tuple belongs to first window
9:  if((tuple.time−qp.start <= w.size and w.size < w.step) or (tuple.time−qp.start <= w.step and w.size > w.step))
    then
10:     return wids.add(1)
    Case 3: w.size>=w.step: tuple belongs to a window
11: if(tuple.time >= maxW.start) then
12:     tuple.maxW = ceiling + 1
13:     wids.add(tuple.maxW)
14: if(tuple.time <= maxW.start) then
15:     tuple.maxW = floor + 1
16:     wids.add(tuple.maxW)
    Case 4: w.size<w.step: tuple belongs to more than one window
17: while(tuple.time <= [(tuple.maxW − 1) ∗ w.step + w.size] and (tuple.maxW ≠ 0)) then
18:     wids.add(tuple.maxW)
19:     tuple.maxW = tuple.maxW − 1
20: endwhile
21: return wids
```

those tuples in its current state which belong to the same windows that the new tuple belongs to. It is possible that a query plan can be composed of multiple stateful operators each having different window characteristics. Figure 2.4 is an example of such a query plan. As is illustrated, while join operators 1 and 2 have defined hopping windows on their outputs with the following characteristics: window sizes of 5 and 4 time units and window sliding steps of 7 and 6 time units respectively, join operators 3 and 4 have defined continuous windows with widow sizes of 8 and 7 time units, and window sliding steps of 3 and 5 time units respectively.

In this work we assume time-based windows. However, the techniques we propose can easily be applied to count-based windows. We assume that a tuple's timestamp is set at the data source and that tuples arrive at the DSMS in order. Network delays and unsynchronized data sources may cause disorder in the incoming data streams. How-

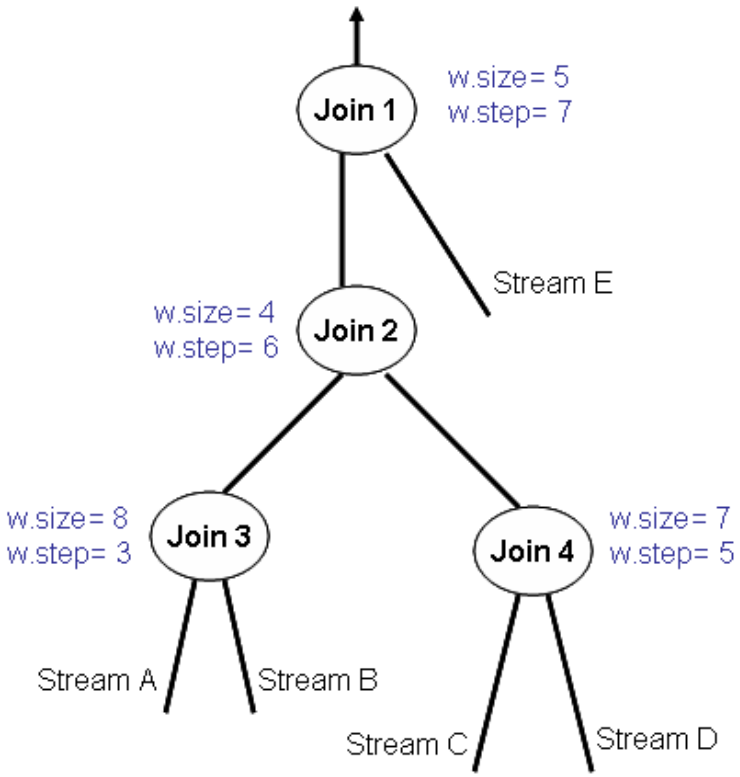

Figure 2.4: Example of a query plan with different window sizes per operator.

ever, different techniques have been already proposed in the literature for resolving such issues. One such mechanism is called heartbeats [33]. Heartbeats are punctuations on the timestamps of tuples which can be generated either by the data sources or by the DSMS. A heartbeat with a timestamp $t_1 = 12:05$ indicates that all tuples with timestamps $tuple.time <= 12:05$ have been received and no more such tuples are expected henceforth. Another mechanism discussed in the literature is called *Slack*. *Slack* allows disorder in the data streams within predefined bounds [33]. The generation and propagation of heartbeats is out of the scope of this work. In our work only the leaf operators of a query plan assume that tuples arrive in order. Since spilling and unspilling data to disk changes the natural order of tuples, we have implemented mechanisms for handling out of order tuples. We employ communication techniques called *punctuation pointers* which

help operators exchange information with their ancestors about the current stage of their data-processing. In Section 2.3 we provide an exact definition of a *punctuation pointer* and an explanation how punctuation pointers are incorporated into our framework.

2.1.1 Query Syntax for Window-Based Operators

The sql syntax does support the definition of queries with time constraints imposed on their output. This has prompted the development of a new query language called CQL [25]. CQL is an SQL based language. It has a rich syntax which allows for the definition of count-based and time-based constraints. CQL, however, does not have a clearly defined and flexible window semantics which would allow the expression of different types of windows in a query. [18] presents such a semantics. Query 1 is an example of a multi-join query defined using a CQL-like syntax. The query assumes the existence of a stream processing financial system which receives financial data from various banks and other financial institutions. The query joins streams over a 10 minute period and outputs the data every 3 minutes.

```
QUERY 1:
  SELECT brokerName, min(price)
    FROM bank1,bank2,bank3
   WHERE bank1.offerCurrency=bank2.offerCurrency
         AND bank2.offerCurrency=bank3.offerCurrency
         AND bank1.offer=bank2.offer
         AND bank2.offer=bank3.offer AND
         bank1.timestamp>=bank2.timestamp+window
         AND bank1.timestamp>=bank3.timestamp+window
GROUP BY brokerName
```

The same query can be expressed also as the query below where *WATTR* stands for window attribute [18]:

```
QUERY 1:
```

```
  SELECT brokerName, min(price)
    FROM bank1,bank2,bank3
   WHERE bank1.offerCurrency=bank2.offerCurrency
         AND bank2.offerCurrency=bank3.offerCurrency
         AND bank1.offer=bank2.offer
         AND bank2.offer=bank3.offer AND
         bank1.timestamp>=bank2.timestamp
         AND bank1.timestamp>=bank3.timestamp
         [WATTR timestamp RANGE 10 minutes SLIDE 3 minutes]
GROUP BY brokerName
```

2.2 Invalidation Rules

As the execution of a query plan proceeds, no longer necessary data accumulates in main memory, thus limiting the availability of memory resources for the storing and processing of new tuples. This necessitates the discarding of any outdated data accumulated in the states of operators. The state of an operator consists of all the tuples which have to be buffered so that the operator can produce complete and accurate results. To be able to ensure correct and complete query plan output, operators need rules for detecting when data will be no longer needed. We provide a set of rules which guarantee that only unnecessary, already processed data will be discarded by operators.

Since windows have finite sizes, once an operator has received and processed all tuples that belong to a given window, these tuples can be discarded granted the condition that they do not belong to other windows the operator is still processing input for. The deletion of no longer needed tuples reduces the overall size of the state of an operator, thus reducing the memory resources consumed by it. In the context of DSMS, the process of detecting and deleting no longer needed tuples by a window stream operator is called *invalidation* [19]. An effective invalidation strategy can help an operator achieve better and more efficient management of its memory resources. To guarantee correct and com-

plete query plan output, which means that no tuples are discarded before all output results for a given window have been produced, we define two invalidation rules.

Invalidation rule 1: *Tuples in an operator's state can be invalidated when they belong to only one window and all the tuples within the window's range have already been received and processed.*

The rule is correct. It needs no further conditions because the conditions are embedded within the rule. If the following two conditions are both satisfied, then rule number 1 alone is sufficient to guarantee that no longer needed tuples are invalidated at the earliest possible moment:

- Condition 1: the window is hopping.
- Condition 2: tuples arrive in order at the operator.

However, if tuples arrive out of order, and the windows defined on the operator are continuous, then rule number 1 is not sufficient to guarantee that only outdated tuples are deleted from an operator's state. Before proceeding with the definition of a second more general rule, several concepts need to be defined.

As it has been already explained, in the case of continuous windows, a tuple may belong to more than one window at a time.We define the degree of window correlation (*dwc*) to be the number of windows that a window shares tuples with excluding itself. *dwc* is equal to the maximum number of windows a window can share tuples with minus one. The maximum number of windows can be calculated using Formula 1.

We note that hopping windows have a degree of window correlation of zero. As no two hopping windows have overlapping boundaries, a hopping window shares no tuples with other windows. This can be seen in Figure 2.2. As another example, consider a continuous window with $w.size = 7$ time units and $w.step = 2$ time units as illustrated in Figure 2.5. The degree of window correlation for this window is equal to 3 as calculated

using Formula 1 and subtracting 1 from it ($\lceil(7/2)\rceil - 1 = 3$). Thus each window for such a query plan shares tuples with at most three other consecutive windows. As shown in Figure 2.5, w1 has overlapping boundaries with w2, w3 and w4. Window 1 ends before window 5 begins. Thus a tuple with a timestamp of 12:06:45 belongs to four windows simultaneously: w1 and the three other windows w1 shares tuples with, namely windows: 2,3 and 4.

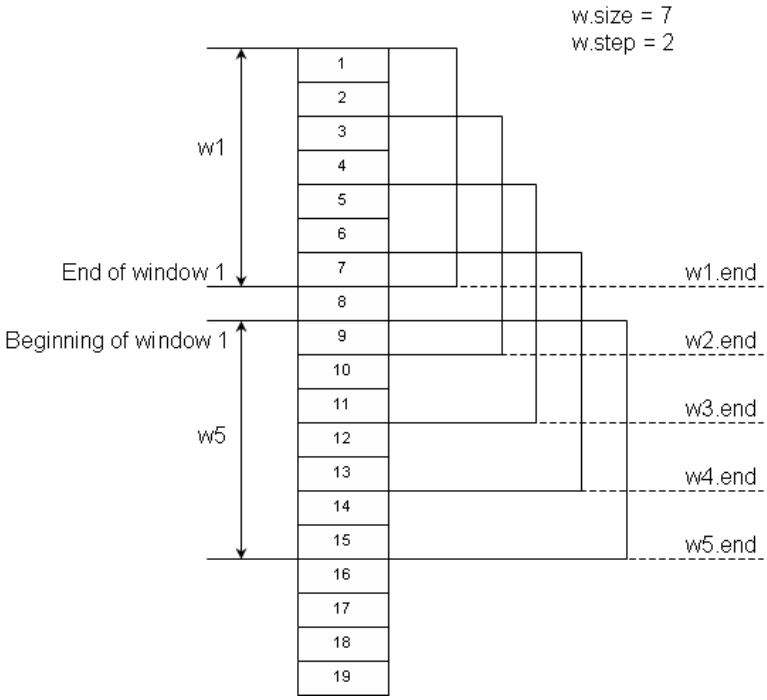

Figure 2.5: Example of a window with $dwc = 3$.

The degree of window correlation determines the number of windows a window may share tuples with. Thus it implies how many consecutive windows should have received and processed all the tuples that belong to them before tuples from the first window can be safely discarded by the operator and the window can be closed. Note that such ”interleaved” windows may or may not have yet received and processed all the tuples that belong to them. Thus an operator cannot safely invalidate tuples from a window even if all the tuples falling within the window’s range have been already received and processed.

Tuples from a processed window may be needed for the completion of the processing of other windows so our goal is to determine a method to decide when it is safe to purge tuples.

Closing a window means discarding all meta data (records) maintained by the operator for the given window. We number windows starting from 1. Thus the most currently opened window will have the largest window identification number. If we start invalidating tuples from the most currently opened window, then we use what we define as *backward window correlation* to determine whether it is safe to discard tuples from this window or not. However if we start invalidating tuples from the earliest open window, let us say, for example, from window 1, then we need to determine how many windows after the earliest opened window have received and processed their tuples. We define this as *forward window correlation*.

Based on how many tuples for a window have been received thus far, a window can be in several different states. Figure 2.6 shows the two main states a window can be in: *open* and *closed*. Each of these states is characterized by two sub-states.

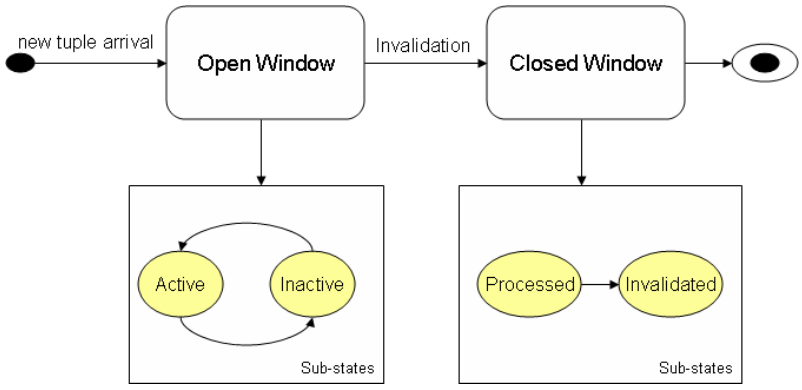

Figure 2.6: A window's state transition diagram.

An *open* window is a window for which at least one tuple has been received by the operator. Yet all tuples belonging to the window have not been processed yet. Based on whether all the tuples received thus far for the window are in main memory or not, an

open window can be further defined as either *active* or *inactive*. An *active open window* is an open window for which at least one tuple is residing in main memory. An *inactive open window* is an open window with no tuples within its range currently residing in main memory. This implies that all thus far received tuples within the window's range have been spilled to disk. Even though an *inactive open window* may currently have a size of 0, the operator may not have produced yet all possible output results for this window. For it is possible that more tuples are expected to arrive at the operator or that tuples temporarily put on disk may still need processing.

We define a window to be a *closed window* if no more tuples will ever be received by the operator from its children- children can be either other operators or input streams. A *closed window*, however, may or may not have its tuples invalidated. Thus, a *closed window* can be further defined as *processed* or *invalidated*. A *processed closed window* is a window for which all output results have been received but no tuples within the window's range have been invalidated yet. This may bedue to the fact that its tuples are still being required for processing of other windows. A processed window has no fragments of it spilled on disk. On the other hand, an *invalidated closed window* is defined to be a closed window with all its tuples invalidated. If any meta information for this window has been maintained by the operator during query plan execution, such meta information can be now safely deleted because by invalidation rule 2 the tuples from this window are guaranteed to be no longer needed for any processing in the future. Invalidation rule 2 is provided below.

To ensure that all requirements necessary for the safe invalidation of tuples have been met in the case of out of order tuples and continuous windows, we define a second invalidation rule.

Invalidation rule 2: A tuple can be invalidated if and only if all the windows the tuple belongs to are in a closed processed state.

The intuition behind invalidation rule 2 is that if a tuple belongs to processed windows only, then all possible output tuples that involve the tuple have been already produced and output by the operator. Furthermore, no future output results will involve this tuple as no newly arriving tuples will belong to the windows the tuple belongs to. Thus the tuple can be safely discarded by the operator to release memory for new data.

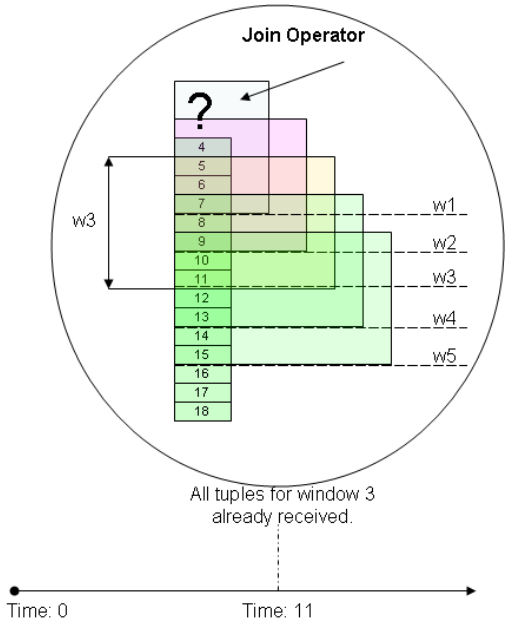

Figure 2.7: Invalidation example with w.size=7 min and w.step=2 min.

For example, let us again assume that the state of a join operator is divided into continuous windows with $w.step = 2$ time units and $w.size = 7$ time units as shown in Figure 2.7. Let's further assume that at time $t.time = 11$ tuples within the time-range $tuple.time = 1$ to $tuple.time = 3$ have not been received yet by the operator as indicated by the question mark in Figure 2.7. Furthermore, the operator has no information about whether tuples with timestamps between 1 and 3 should be expected. If any tuples within the range $tuple.time = 1$ to $tuple.time = 3$ arrive at the operator they will belong to w1 and w2. All tuples with a timestamp greater than 3 have been arriving at the operator in consecutive order. Thus at time $t.time = 11$ the operator has received the last tuple

within w3's range. As is illustrated in Figure 2.7, w3 starts at $t.time = 4$ and ends at $t.time = 11$ and it shares tuples with w1 and w2. By calculating the degree of window correlation with the help of formula 1, we can verify that w3 should share tuples with three other windows. As w3 is the third consecutive open window, it can share tuples with 2 other windows only- namely w1 and w2. Since w1 and w2 are still expecting new tuples, no tuples which belong to either of these windows and to w3 at the same time can be invalidated. The last tuple which belongs simultaneously to w2 and w3 has a timestamp of $tuple.time = 9$. Given these conditions at $t.time = 11$, can we invalidate any tuples from w3? Tuples with timestamps t.$tuple.time = 10$ and $tuple.time = 11$ do not belong to any of the previous windows: w1 and w2, however, they do belong to w4 too, besides w3. At time $t.time = 11$ w4 will be in an *open, active state*, therefore no tuples which belong both to w3 and w4 can be invalidated either. Thus at time $t.time = 11$ no tuples can be invalidated.

The check if an invalidation process can start can be triggered either by the expiration of a predefined time interval (time-driven invalidation) or it can be triggered by the finalization or the beginning of an expected event (event-driven invalidation). One example of a plausible invalidation check trigger would be the expiration of a sliding step period as the end of each sliding step marks the end of an already open window and the beginning of a new one.

The invalidation process in our system, however, is a combination of both triggers- it is time- and event-driven. We use an invalidation interval bigger than a window's sliding step. As we assume that tuples may arrive at an operator out of order, the end of a sliding period would not necessarily mean that all tuples for the just expired window would have been already received and processed by the operator. Tuples from this window may have been spilled to disk either locally or upstream the query plan [1], that is, at

[1]Data in a query plan flows from leaf operators to the root. Thus, the direction from the leaves to the root of a query plan tree is called downstream and vice versa from the root operator to the leaf operators

children operators. Thus by using an invalidation timer longer than a window's sliding step valuable processing resources will not be wasted by frequent invalidation checks at the end of each sliding period, as each invalidation attempt causes some overhead.

In our system we use two more event triggers: tuples are invalidated before a spill process starts and after an unspill process is finished. These triggers are independent of the adaptation policies used. The intuition behind setting these triggers is the following. Before an operator spills tuples to disk, the operator can check if any tuples can be invalidated from its state as this could reduce the size of data written to disk, therefore reducing the cost of spilling. Moreover, the time necessary for bringing the same data back to main memory will be also decreased. Thus, by doing an invalidation check before a spill process, we anticipate that valuable computing resources may be saved in the long run. As some windows may not be invalidated because parts of the windows have been spilled to disk locally before joining them with later arrived tuples within the same windows' ranges, whenever tuples are unspilled these windows may have all tuples within their bounds received and processed. Thus, the tuples from these windows can be invalidated. By invalidating at the end of an unspill process we anticipate to achieve more efficient memory management. As the end of an unspill process marks a spike in the memory consumption of the operator anyway, the invalidation of any no longer needed tuples will reduce the amount of consumed memory at that moment.

2.3 Invalidation Synchronization Mechanism

As we assume that query plans may consist of multiple state-intensive operators which may result that some tuples be out of order at non-leaf operators, the timestamps of the tuples alone do not provide enough information for making a correct decision regarding

upstream.

what tuples can be safely invalidated from an operator's state. We assume that tuples arrive in order at leaf operators so the timestamps of the tuples themselves can provide enough information for the invalidation process at these operators. Out of order tuples downstream the query plan can be caused by spill adaptation processes executed upstream the query plan. Thus operators need a mechanism of synchronizing their invalidation processes with the processing stages of operators located upstream the query plan. By synchronizing the invalidation processes we mean that no operator can invalidate tuples from its state, unless it has been explicitly notified by all of its children operators that no tuples with timestamps within the range of the tuples to be invalidated will be received in the future. A simple yet elegant way to achieve such synchronization is by using punctuated tuples (or also called punctuated messages) interleaved with the data stream.

Punctuation is a way of inserting meta-data about the data stream by encoding the information into special-purpose tuples. [19] defines punctuation as "an ordered set of patterns, each corresponding to an attribute of the tuple". We overload the punctuation message with extra information about the stage of processing which has been just completed by the sending operator. In our system we use punctuated messages to inform operators downstream about the time-ranges of tuples that have been already processed upstream. Thus downstream operators will know that tuples within such data ranges will never come again. This gives them information to make correct invalidation decisions. The punctuated messages used by our invalidation policy are called *invalidation pointers*. As it was discussed, an *invalidation pointer* contains the time ranges of tuples that have been already processed and invalidated upstream. To guarantee correct invalidation information, an operator is allowed to send invalidation pointers only after it has successfully invalidated tuples from its state. As we assume in-order arrival of tuples at leaf operators only, which are directly connected to the data sources, leaf operators do not receive explicit punctuation pointers. Such operators thus initiate them. An implication of this is

that leaf operators should be able to invalidate some tuples from their state at the expiration of each sliding step. This may not be a very efficient strategy though if the windows defined for an operator are large, continuous windows with very small sliding steps.

2.4 Granularity of Spill Units

In our work we consider query plans composed of multiple state-intensive join operators. The state of an operator corresponds to input tuples an operator has to keep buffered in order to produce accurate and complete query results. In the streaming context the size of the state of an operator is implied by the characteristics of the windows imposed on it. For example, hopping windows with smaller window sizes imply smaller operators' states. As windows of state-intensive window operators may grow too large due to spikes in input which may cause strain on machine resources, adaptation techniques at the state or state-window level of an operator such as state spilling to disk may become very expensive. To overcome the problem of potentially too costly query optimization strategies, the state of a stateful operator can be partitioned into numerous non-overlapping subsets of tuples. Thus during a query optimization stage only a subset of the whole operator's state such as window fragments may be potentially locally spilled to disk. In our work, we partition each input stream into a large number of partitions per window, as proposed in [11]. Every operator partitions the data as it comes, using its own partitioning function. Each partition is identified by a unique partition ID, i.e., 1, 2, . . ., n (with n denoting the number of distinct partitions). This gives us the opportunity to effectively work with partitions during an adaptation process without even having to rehash any of the existing ones at run time. This method has first been applied in early data skew handling literature [15] as well as in the recent stream processing work Flux [27]. We organize operator states based on input partitions. For simplicity, we may use the term partition to refer to the corresponding

operator state window partition if the context is clear.

For a single input query operator, as tackled in Flux [27], it is natural to adapt partitions from this one input stream. However, as discussed in [10], for the multiple-input operators we focus on, there are partitions from different inputs in the operator states with the same partition ID. Thus, multiple ways of organizing partitions are possible, as discussed below. As in XJoin [32], we could choose partitions from one input at a time and adapt them independently. However, this strategy increases the complexity of bringing back to main memory any temporarily flushed to disk tuples in the partitioned processing of multi-way join queries. Namely, if partitions have been pushed to disk, the operator will be required to keep track of extra timestamps per tuple and per partition to avoid duplicates later when spilled data is brought back to main memory. Thus we instead use the [9, 10] idea of synchronized flushing of a group of partitions with the same partition ID, but we take the same partition group per window. This is used as the smallest unit to be adapted, as illustrated in Figure 2.8. This simplifies the unspilling process.

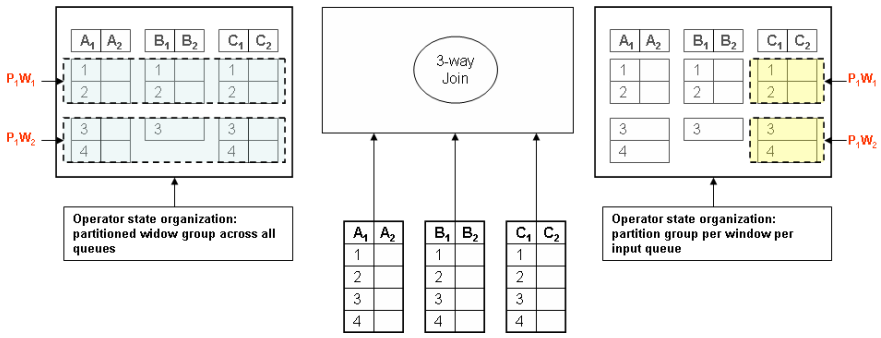

Figure 2.8: Example of a partitioned window group.

For simplicity, we call all partitions of a window with the same partition ID from different inputs one *partition group*. The term partition may be used to refer to a window partition group if the context is clear. The processes of spilling and unspilling data to disk will be described in more details in Section 2.6 .

Granularity of Adaptation. The number of partitions into which the windows of an operator are divided determines the granularity of the adaptation level. Thus controlling the number of partitions provides control over the costs incurred during a query optimization process. More partitions would imply smaller partition groups, therefore smaller costs when data is written to disk or read back to main memory. Too small units, however, may cause too many writings to and readings from disk, thus increasing the overhead of adaptation. One has to find the balance based on the system's characteristics.

2.5 Partitioned Window Join Operator

By dividing the state of an operator into smaller independent adaptation units, more cost efficient optimization strategies can be applied at the state level of an operator. We call these adaptation units partition groups by window (*partitioned window group*). An operator with its state split in partitioned window groups is called *Partitioned Window Operator*. In this work we focus on the design of a Partitioned Window Join Operator due to frequent usage of joins in query plans.

The design of a Partitioned Window Join Operator has to be flexible enough to allow the swapping of states between disk and main memory upon demand while still outputting complete and accurate query results. No output tuples should be missed nor should extra data be generated during query processing. Since operators within a query plan affect each other's work due to the nature of the query processing, the swapping of tuples between disk and main memory by an operator at a higher level [2] of the query plan ultimately affects the order in which lower level operators receive tuples. Thus a *Partitioned Window Operator* ought to be able to process out of order tuples while still guaranteeing accurate and complete results.

[2] In this work we start labeling the levels of operators in a query plan tree starting from the root operator. The root operator is located at level 0, its children operators and level 1 and so forth.

As it was already described in Section 2.3, the processing of out of order tuples requires the presence of an information exchange mechanism among operators in the query plan. If no such information exchange technique exists, an operator will have no means of knowing when the last tuple of a window for a specific partition group has been received. Thus, no longer needed tuples cannot be properly invalidated to release memory for new data and the system processing the query plan has a very high probability of running out of memory. To prevent this, we employ the use of punctuation pointers as described in Section 2.3. Thus, every partitioned window join receives and sends out punctuation pointers. The information in the punctuation pointers is organized in time ranges by partition group ID. Every operator keeps track of this information as it is needed during data invalidation.

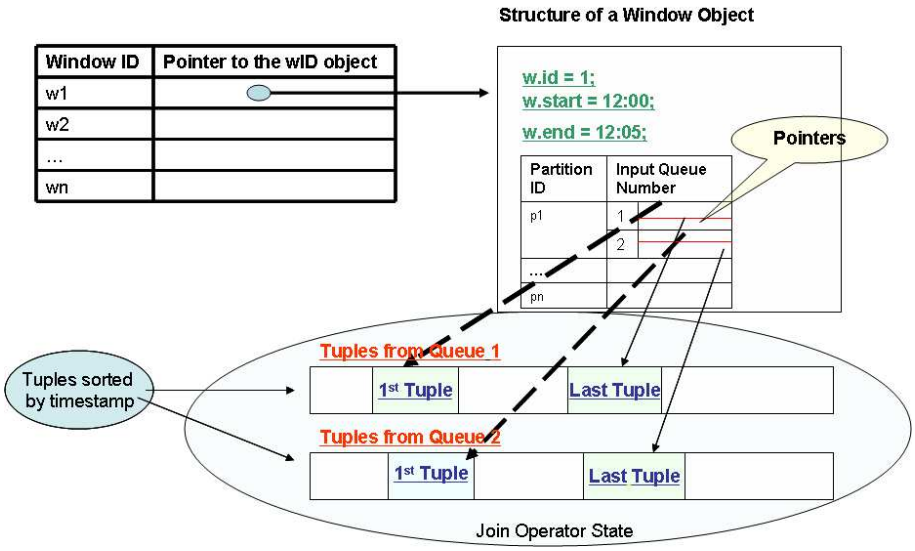

Figure 2.9: Window map diagram.

Another design consideration is the per window organization of the tuples within the state of an operator. As main memory is a critical resource that has to be managed very efficiently in the streaming environment, having data tuples duplicated across overlapping windows is not desirable. Therefore a partitioned window operator needs a data structure

which would allow the management of window information per partition group without the need of duplicating tuples across overlapping continuous windows. Tuples need to be processed on the fly and only one copy of each tuple should be stored. Such design considerations are used in the design of the aggregate window operators using window semantics [18].

As shown in Figure 2.9, this data structure has to contain meta-data about every window such as beginning and ending time of the window, state of the window per partition as shown in Figure 2.6, pointers to the tuples which belong to this window. This data structure has to be updated every time new tuples are inserted into the operator's state or old tuples are invalidated, or tuples are swapped between disk and main memory. We call this data structure a *window map*.

2.6 Definitions of Spilling and Unspilling

In this work we focus on the design of query optimization policies which swap portions of the states of an operator between disk and main memory as needed based on the availability of system resources and data arrival rates. We call the process of flushing a portion of the operator's state to disk *spilling* and we call the process of bringing back to main memory data spilled to disk *unspilling*. By swapping data between disk and main memory based on the characteristics of the data rates of input streams, more efficient memory management can be achieved. By spilling partially processed data to disk during periods of high data arrival rates, we prevent the system from crashing while still producing complete and accurate results, though at the expense of a delay in outputting the complete query result.

An implication of swapping data between disk and main memory is the need of keeping extra information about the data being flushed to disk. This information is needed to

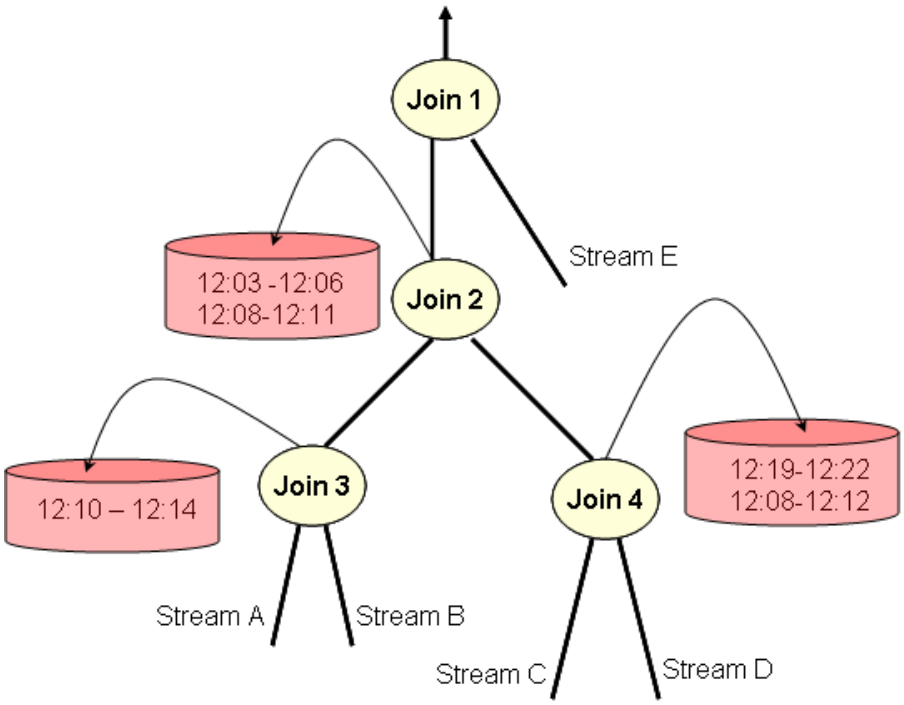

Figure 2.10: Spilling example.

help prevent duplicate query results. As it was already described in Section 2.4, we use a partition group as the smallest adaptation unit. Thus we avoid having to keep concurrent time information per spill. This reduces the complexity of bringing back this data to main memory. We follow the rule that no tuples spilled to disk have ever been joined with any data residing in main memory. All tuples spilled to disk have been joined with each other. We do keep extra statistics per spilling such as number of tuples spilled, partition group id from which tuples are flushed to disk, timestamps of the first and the last tuple spilled. This statistics is used by our unspilling policies when deciding which data to bring back to main memory first.

Spilling and unspilling data poses many interesting questions such as:

- How much data to un/spill?
- What data to un/spill first?

- How often to un/spill?
- When to un/spill?

In this work we look at all the question but our main focus are the first and the second questions, selecting which windows and partitions to un/spill first while achieving maximized query plan throughput given the current state of the system. There are different interconnections between operators and the states of these operators in a multi-operator query plan. We exploit these interconnections to achieve more synchronized query plan processing. This will be discussed later when we describe the spilling and unspilling policies designed for our framework.

Figure 2.10 illustrates the impact of spilling on the order in which operators located at lower levels of the query plan such as the root receive tuples from their children operators. The figure illustrates what happens when an upstream operators spills data to disk. At time $t.time = 12:22$ the root operator may have received all tuples with $tuple.time <= 12:21$, however the root operator still expects tuples with such timestampsto come from its child operator. As the figure illustrates, tuples with timestamps within the ranges of 12:03-12:06, 12:08-12:14, and 12:19-12:22 have been spilled to disk upstream and are still to be processed and sent down the query plan. Thus the tuples arrive no longer in order at the root operator.

As illustrated on Figure 2.10 the ability to spill and unspill data on demand affects also the invalidation process. Tuples have to be kept in the state of an operator until all query results involving these tuples have been produced. Spilling tuples upstream prevents the affected windows from being invalidated solely based on the timestamps of the tuples, as the timestamps are no longer a valid indicator of the order in which tuples have been received by the DSMS.

Invalidation in the context of spilling and unspilling. Information such as the par-

tition ids and the timestamps of the tuples affected by local adaptation processes such as data spillings or unspillings need to be taken into consideration during tuple invalidation. If an operator has received all tuples within a window's time range but a portion of this window has been spilled to disk locally, then such an operator has not yet produced all result tuples for this window. Therefore, before all tuples residing on disk and falling within this window's range have been brought to main memory and processed, no tuples can be invalidated from this window.

As it has been already explained in Section 2.1, hopping windows have no overlapping boundaries. Thus spilling tuples from one window locally does not impact other currently open windows. This simplifies the invalidation process. Once an operator has received and processed all the tuples within a window's time range, given that no tuples have been locally spilled to disk, the operator can invalidate tuples and close the window, irrespectively of whether earlier windows have been still open because of tuples locally spilled to disk or tuples spilled to disk upstream. However, this is not the case when the state of an operator is divided into continuous windows due to the higher than 0 degree of window correlation. Thus in the case of continuous windows, when the operator has received and processed all the tuples falling within a window's time range, the operator cannot invalidate tuples from this window before making sure that no tuples contained by the window are still needed by earlier, open windows. Invalidation punctuation pointers received from upstream operators provide information about what time ranges it is safe to invalidate tuples from. Spill punctuation pointers carry information about data spills that have taken place at upstream operators. The information in both types of punctuation pointers is organized by partition group ID and the time ranges of tuples either invalidated or spilled upstream. Operators also maintain data about the partition groups and the time ranges of tuples locally spilled to disk. Thus, by using these time ranges and metadata about the windows, an operator can calculate the range of windows affected by

any upstream spills and further infer the timestamps of the tuples that is safe to invalidate.

Given locally spilled tuples, Formula 3 can be used to calculate the earliest received tuple that can be invalidated from a given window. The formula uses the information from Table 2.1.

Formula 3: $tuple.time = maxSpilledW.end + [w.step - ((dwc + 1) * w.step - w.size)]$+1

Variable name	Description
maxSpilledW.end	This is the end time of the latest window affected by the spill. For example, if tuples with timestamps ranging from 12:04 to 12:06 have been spilled to disk and the tuple with timestamp 12:06 belongs to windows 2 and 3. The latest window affected by the spill will be window 3.
dwc+1	This is the maximum number of windows a tuple could be shared by given a window size and a window sliding step. The maximum number of windows is equal to the degree of window correlation (*dwc*) plus 1.

Table 2.1: Variables used in the invalidation algorithm.

Algorithm 2 Tuple Invalidation Algorithm.

```
1: Hashtable timeRanges = calculateSafeToInvalidateTimeranges()
2: List spilledWindows = getSpilledWindows()
3: markProcessedWindows(spilledWindows,timeRanges)
4: List openWindows = getOpenWindows()
5: for (every wi in openWindows) do
6:     int dwc=(⌈w.size/w.step⌉ − 1)
7:     boolean canInvalidate=true
8:     while (dwc >= 1) do
9:       if (!wi.state.equals(closed)) then
10:          canInvalidate=false
11:    endwhile
12:      if (canInvalidate) then
13:        invalidateTuples(wi.start, w.step)
14: endfor
```

Algorithm 2 outlines the major steps of the invalidation process. This process takes into account information about windows and partitions locally spilled to disk and information about windows and partitions already processed upstream.

First, the algorithm calculates what time ranges are safe to invalidate tuples from. Information provided by the synchronization punctuation pointers- spill and invalidation punctuation pointers, information kept by operators about tuples locally spilled to disk, as well as the timestamps of the latest tuples received from any direct input streams are used when calculating these time ranges. The second step is checking the windows' bounds

and deciding which windows have already received and processed all the tuples that fall within their limits. The states of such windows are marked as *processed*. The last step of the algorithm looks at all the windows whose states have been marked as *processed* and based on the states of all the windows these windows share tuples with, the algorithm either deletes tuples from them and changes their state to invalidated so that they can be closed later, or it leaves their status unchanged, deleting no tuples from them. Windows marked as invalidated can be closed. This means that any meta information kept by the operator about them can be safely deleted. The last step of the algorithm uses the degree of window correlation to determine how many windows back it has to look into before it decides whether tuples can be invalidated or not.

2.7 Content-Based and Time-Based Interconnections in a Query Plan

As described in Section 2.4, the state of each partitioned window join operator is organized by partition ids and window ids. Thus two levels of dependency (correlation) can be observed between partitioned window join operators located at different heights of the query plan tree:

- **content-based dependency**-considers the partition groups a tuple belongs to as it moves downstream the query plan. Thus, this is the correlation among partition groups of operators located at consecutive levels of the query plan tree.

- **time-based dependency**-considers the window ids a tuple belongs to as it moves downstream the query plan. Thus, this is the correlation among windows of operators located at consecutive levels of the query plan tree. Operators in a query plan can have completely different window characteristics.

As it was investigated in [10], a many-to-many relationship exists between partition groups. As Figure 2.11 shows, many tuples from partition 1 at operator one may be potentially hashed to partitions 1 and 2 at operator two. Thus the spilling of tuples to disk from one partition group upstream may potentially affect many partition groups downstream. The same holds true for unspilling.

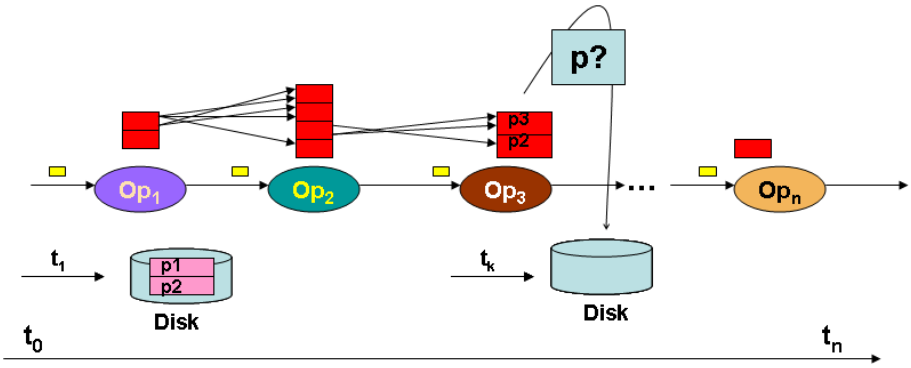

Figure 2.11: Example of content-based dependency between operators.

The same type of correlation is observed regarding the association of tuples to windows. Since at each operator of the query plan tree a tuple will be associated with one or more windows and since no window can be considered closed unless all tuples within the window's range have been received and processed, spilling tuples to disk from one window upstream will affect the state of the correlated windows downstream. Thus, since windows of operators located downstream will not be closed and invalidated unless all tuples from the correlated windows upstream have been unspilled first, adaptation strategies for partitioned window join operators need to account for these interdependencies among operators in a query plan to achieve optimal query plan processing performance.

Operators need to synchronize with each other the adaptation stages they complete to achieve optimal performance since they are not completely independent units within the query plan. The same intuition holds for the construction of production lines in a manufacturing plant. The work of each station along a production line has to be synchro-

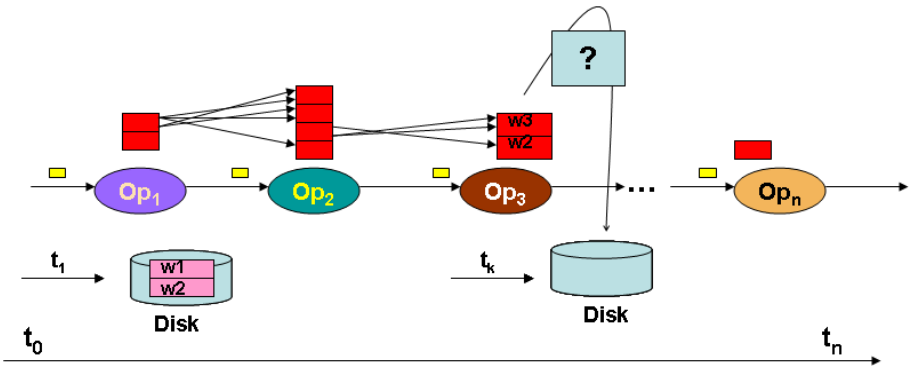

Figure 2.12: Example of time-based dependency between operators.

nized with the work of the previous stations, otherwise no useful work will be achieved. In a chocolate production line, for example, unless the chocolate mixture has been prepared, the station responsible for pouring the mixture in the chocolate shape forms will be blocked. Unless the chocolate has been cut and put in the correct shapes the next station along the production line where wrapping occurs will be blocked. Unless the chocolate bars are properly wrapped, the station where the bars are packed in packages and prepared for shipping out of the company will be blocked. A query plan tree is similar to a production line. Each operator can be viewed at as a separate station across this production line, since every operator provides the input queues for its parent operator and depends on the output queues of its children operators. Thus our intuition is that a better synchronization in the work of operators will provide for optimal query plan performance.

Chapter 3

Policy Design

3.1 Local Policies versus Global Adaptation Policies

By spilling and unspilling data on demand, a data stream management system can satisfy the requirements of applications for complete and accurate query results, while preventing a system crash during periods of high load and a waste of resources during periods of light load. As is well known, to achieve optimal performance, every adaptation decision needs to be fine-tuned based on the current state of the system reflected by the values of collected statistics. Based on the answers to the most important adaptation questions such as the ones listed below, different adaptation scenarios and policies are possible.

1. When to start adaptation?
2. How much data to use during each adaptation step?
3. What should be the smallest data unit size we should work with?
4. How to select what partition groups and windows to use during an adaptation step?

In our work, we divide the adaptation polices we have designed into two major groups based on the type of statistics each policy uses. The two groups are: *local adaptation*

policies and *global adaptation policies*.

Local policies use statistics which reflects the status of only a sub-part of the whole system (query plan). For example, let us assume that the final goal of all the adaptation processes is to achieve maximal throughput. We define as throughput the number of tuples output by the query plan for the whole time the query plan has run thus far up to time *t*. Let us further assume that the statistics collected by each operator is per partition group.

As shown in Figure 3.1, each operator collects data about the size of each partition group (number of input tuples) and the output rate of the group (number of output tuples per unit of time) so that it can keep track of the most productive and least productive partition groups. We define the productivity of a partition group as the ratio of the number of tuples outputted by the partition group and the size of the partition group. A higher ratio indicates higher productivity [9]. Thus, during periods of high data rates the operator can spill to disk the least productive partitions while keeping in main memory the most productive ones. This would minimize the impact of spilling on the query plan throughput. Vice versa, during periods of low data rates and given that the system has enough free main memory, had any data been spilled to disk, the operator can unspill the most productive partition groups first.

As can be seen in Figure 3.1, each operator updates the measurements of its partition groups independently of the rest of the operators in the query plan. Thus, in this example, the statistical data collected by each operator reflects the productivity of each partition group relative to the operator itself. In other words, the most productive partition group for an operator may be the least productive one relative to the final output of the whole query plan. This may happen, for example, if tuples from this partition group get dropped at operators located downstream the query plan or if they produce less output tuples at downstream operators. Thus local adaptation policies are completely unaware of the overall state of the system or of any inter-correlations that may exist among the

different components (operators) of the streaming system. One of the advantages of local policies is that they are simplest to implement, thus incurring little overhead. Local policies work best for query plans with a single stateful operator. Local policies are discussed in XJoin [32] and Hash-Merge Join [23].

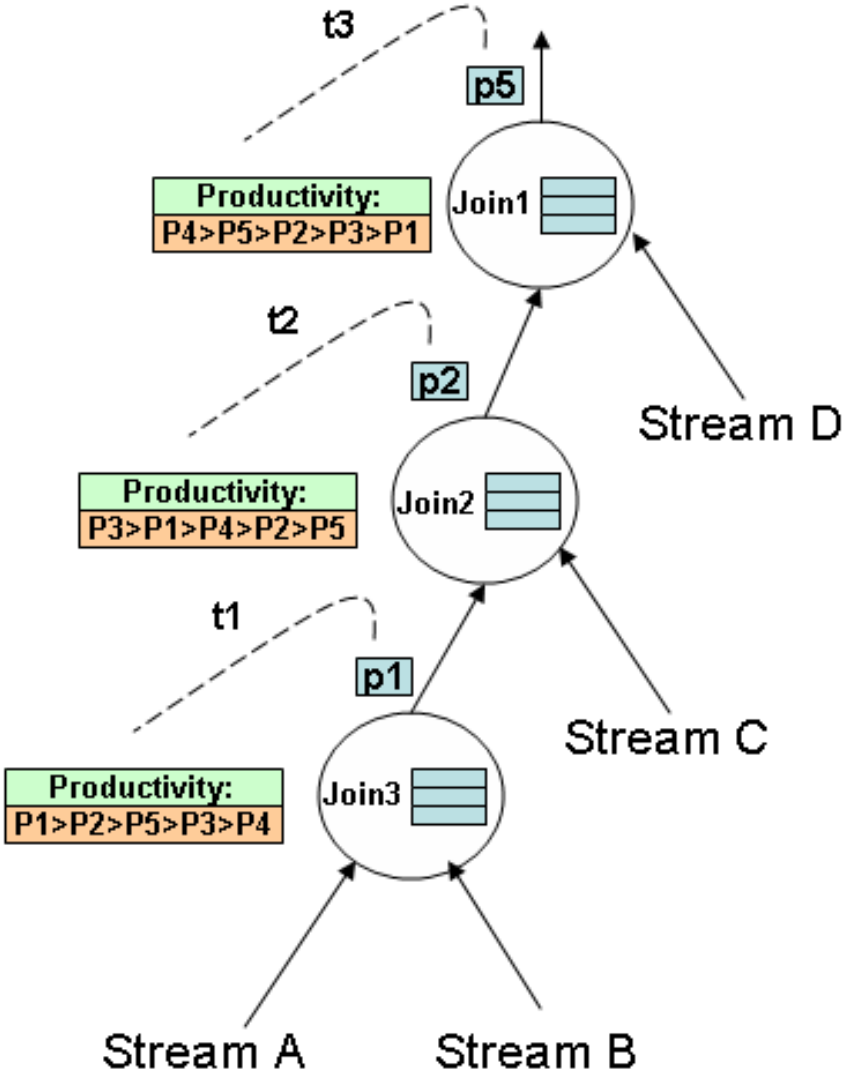

Figure 3.1: Example of a local adaptation policy.

3.2 Global Policies

On the other hand, decisions of global polices are based on statistical data which reflects the overall state of the system since it tries to capture any dependencies which may exist between the different operators in a query plan and their states. Let us go back to our example and assume the same final goal for the adaptation process: maximal throughput, and the same measurements collected by each operator except for one difference. Instead of having each operator update the statistics it collects locally independently of the rest of the operators in the query plan, we have each operator update its measurements only when a tuple is outputted by the query plan. When a tuple is outputted, every operator updates the partition group statistics it collects for the partition group to which the output tuple belonged to when it was passing through this operator. Thus all partition groups to which the tuple has belonged to while passing through operators of the query plan are traced back and their productivity measurements are properly updated. Thus, each operator knows which partition groups are most productive relative to the final output of the query plan. So during periods of system overload, each operator will spill to disk these partitions first which are least likely to contribute to the final output of the query. Thus global policies are more likely to achieve higher throughput than local policies. For further details on the statistical data collected in our system refer to [9].

When comparing adaptation policies, an interesting question is whether there exists a reverse proportional relationship between the number of adaptations triggered by a policy during query plan execution and the query plan throughput. According to [9] a better policy is not necessarily the one triggering fewer adaptations or fewer IOs.

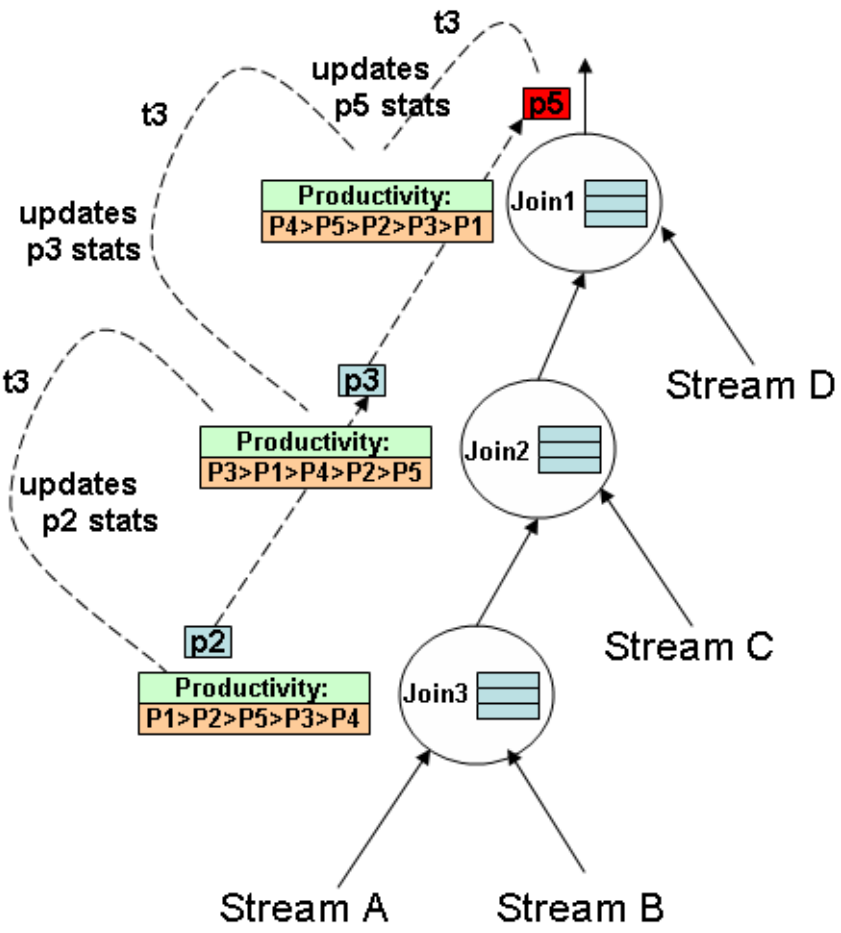

Figure 3.2: Example of a global adaptation policy.

3.3 Spilling Policies

As it has been already explained in Section 3.1, a good adaptation policy is a policy which achieves an optimal system performance while incurring minimal overhead. In our work we measure system performance as number of tuples outputted over certain time. Thus the goal of our policies is to achieve a maximal throughput under different types of load conditions. We have designed five types of spilling policies. Each spilling policy has a complementing unspilling policy. Thus, we group in the same adaptation scenario a pair of matching spilling and unspilling policies to achieve optimal query plan performance.

The five policies we have designed are:

1. **Random Policy**
2. **Local Policy**
3. **Global Unsynchronized Policy**
4. **Global Synchronized Policy**
5. **Semantic Policy**

The most expensive part of our adaptation policies is the actual reading and writing of data to disk. As is well known, access to secondary storage is slow and tends to consume a lot of system resources. In fact, most policies discussed in the literature like flush-largest partition first or flush-all-policy [23], which flush partitions to disk as a way of handling periods of high system aim at minimizing the number of I/Os as this reduces the policy's overhead. However, as discussed in [23], a minimized number of I/Os does not necessarily guarantee maximal query plan throughput. Whenever a partition is flushed to disk, outputting result tuples for this partition is delayed. As new tuples arrive there will be fewer tuples in main memory. This decreases the probability of a join between tuples. Thus in our policies we do consider the number of potential I/Os when deciding what partition groups to spill to disk first, however, this is not the major factor in selecting these partition groups. As discussed in Section 3.1, the statistical data collected by our system which we define as *productivity per partition group* reflects both the size of the partition group and the probability of this partition to produce join results. Furthermore, in an attempt to minimize the impact of our optimization policies on the query plan throughput, in our synchronized policies we try to exploit the content-based interdependencies among the operators in a query plan.

The Random Spilling Policy was designed to help us compare the effectiveness of our policies and the overhead they incur. The Random Spilling Policy randomly selects partition groups for spilling, thus ignoring the size and the productivity of the partition groups and ignoring any time- and content-based operator interdependencies. As compared to other policies, this policy is the cheapest to implement since it incurs no statistical overhead and very low computation overhead. However, as the Random Spilling Policy is blind to the number of I/Os it incurs and the productivity of the partition groups it spills, it is expected to perform worse than the rest of the policies.

The Local Spilling Policy we have designed makes decisions regarding what partition groups to spill based on the localized statistics collected by each operator. To maximize throughput, the policy spills first the least productive partitions. Thus as new tuples arrive they have higher probability of being joined with tuples residing in main memory. This increases the probability of achieving higher throughput. As it has been already explained, though, a problem with this policy is that the collected statistics does not accurately reflect the productivity of the partition groups on a query plan level. The policy also ignores any interdependencies existing between operators and the states of operators in a query plan.

The Global Unsynchronized Spilling Policy in our system takes into consideration globally collected statistics regarding the productivity of the operator's partition groups and the size of intermediate results produced by the partition groups. The Global Unsynchronized Spilling Policy selects the least productive partitions and spills to disk the whole state of these partitions. We use the same formula used by the Global policy with penalty in [9] to calculate the productivity of an operator. As shown in Figure 3.3, this policy, however, does not exploit the existing inter-connections among operators and the states of operators in a query plan tree. Thus, tuples in the states of downstream located operators which belong to partition groups and windows spilled upstream cannot be invalidated until all tuples have been unspilled and processed upstream. Thus, main memory

is not managed in the most efficient way.

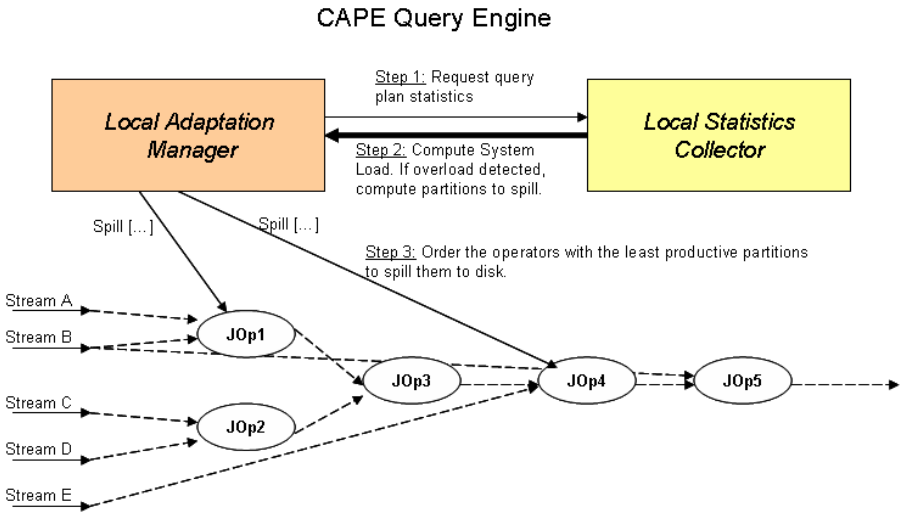

Figure 3.3: Global Unsynchronized Spilling Policy.

The Global Synchronized Spilling Policy is designed to provide for more efficient memory management than the Global Unsynchronized Spilling Policy and the Local Spilling Policy as it is more aware of the intricate connections which exist within a query plan tree. This optimally synchronized global policy uses the same globally collected statistical data as the Global Unsynchronized Spilling Policy. However, as shown in Figure 3.4, the spilling of partitions at operators is done in a synchronized way across the whole query plan. Whenever partition groups for certain windows are spilled to disk at an operator, operators at the next level will spill to disk the partition groups affected by the upstream spilling process.

The spill punctuation pointers carry information about what partition groups and time-ranges of tuples have been spilled upstream. Statistics, locally collected by each operator, keeps track of the correlation existing between its partition groups and the partition groups of its children operators. The locally collected statistics and the data provided by the spill punctuation pointers provide the necessary information an operator needs to decide on

the partition groups that will be most likely affected by the usptream spill process. Thus such partition groups are also spilled to disk. In this way, operators do not have to keep in main memory tuples that first cannot be invalidated because of dependency on spills which have occurred upstream and second have smaller chance of being joined in the near terms, ie, with newly arriving tuples.

To achieve such a level of adaptation synchronization, we use the punctuation pointers described in Section 2.3 to inform parent operators about the partition groups that have been spilled to disk upstream. The beginning of the spilling process will be controlled by a Local Adaptation Manager, located at each query processor. The Local Adaptation Manager will decide which partition groups should be spilled first and by which operator. After this, the spilling processes at parent operators will be triggered by the spilling pointers sent by children operators. Thus a whole chain of spill processes will be spawned which will stop at the root operator. The initially selected partition groups will be the least productive ones across the whole query plan. The initial selection is based on the partition group productivity statistics collected by each operator, as described in Section 3.2.

The Semantic Spilling Policy assigns a different weight per partition group based on information provided by the query plan administrator. The pre-assigned weights indicate the relative importance of the results produced by these partition groups for the final application. Partition groups with less important values will be assigned lower weights. During periods of high data rates and high system load the Local Adaptation Manager will select for spilling to disk first the least productive partition groups which have been assigned lower weights. Thus the more important partition groups remain in main memory and keep producing results. This increases the utility of the streaming system for the final application. A modified formula is used to measure throughput for these policies.

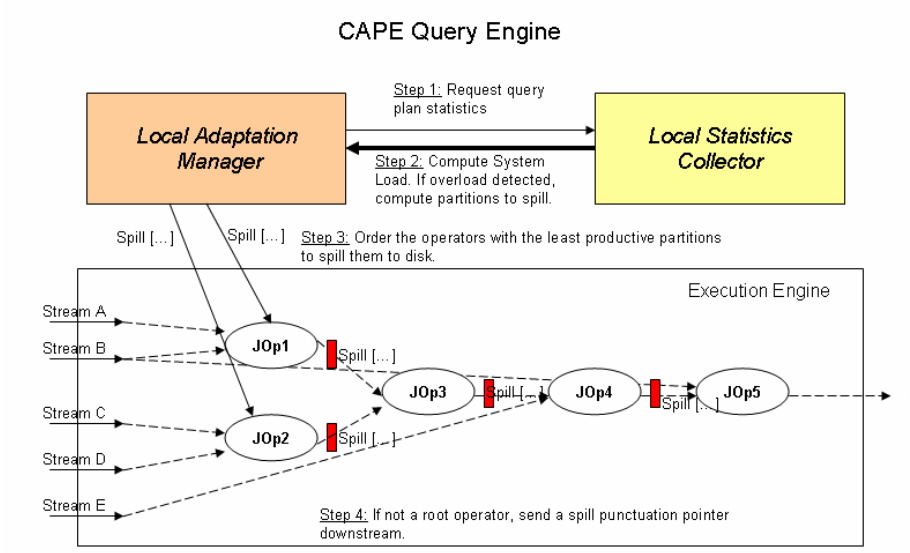

Figure 3.4: Global Synchronized Spilling Policy.

3.4 Unspilling Policies

The unspilling policies discussed in this work use the same concepts as defined in Section 3.3. However, as each policy aims to keep on disk the least productive partitions, during periods of low system load and low data arrival rates the unspilling policies choose to unspill first the most productive partitions. Thus the least productive partitions are always kept on disk if the system cannot process all tuples it has received so far.

The Random Unspilling Policy is similar to the Random Spilling Policy. It randomly selects partition groups for unspilling, thus ignoring the size and the productivity of the partition groups and ignoring any time- and content-based operator interdependencies. All partitions currently spilled to disk have an equal chance of being selected for unspilling. Thus the currently most productive partitions or the biggest partitions spilled to disk, which if unspilled first will incur the least number of disk reads, are not necessarily the ones brought to main memory first.

The Local Unspilling Policy uses the same localized statistical data as the Local Spilling Policy. However, instead of selecting the least productive partitions first, the policy selects the most productive partitions, as the goal is to bring to main memory the partition groups which are expected to produce most output tuples. Thus we still aim to achieve maximal query plan throughput at all times.

The Global Unsynchronized Unspilling Policy uses no content- or time-based synchronization information during the decision process of selecting which partition groups to unspill from disk first as this policy is similar to the Global Unsynchronized Spilling Policy. This unspilling policy uses the same globally collected statistics to decide on the most productive partitions currently spilled on disk which can be unspilled first.

The Global Synchronized Unspilling Policy uses global statistics and considers the content-based interdependencies which exist among operators and the states of operators in a query plan when selecting partitions for unspilling. Similar to the Global Synchronized Spilling Policy the unspilling of partitions at operators is done in a synchronized way across the whole query plan. Whenever partition groups for certain windows are brought to main memory by an operator, punctuation pointers are sent downstream. Upon receiving such pointers, operators at the next level start unspilling from disk the partition groups and windows affected by the upstream spilling process. Statistics, locally collected by each operator, helps an operator to keep track of what partition groups need to be unspilled when such punctuation pointers are received. Thus the processing and invalidation of windows can be optimized.

Unspilling does not have to start at leaf operators only. Any operator of the query plan can be selected initially.

The Semantic Unspilling Policy similar to the Semantic Spilling Policy, uses the weights assigned to the partition groups to decide on which partitions to bring to main memory first. Partition groups with higher weights and higher productivity have a priority

during the unspilling processes. Thus the least productive and least important partition groups remain on disk providing an opportunity for processing more important and more productive partition groups first. As already explained, this increases the utility of the streaming system for the final application. The unspilling policy uses global statistics when calculating the productivity of the different partition groups.

Chapter 4

System Architecture

Our experiments have been conducted on a stream processing system written in Java, which we call CAPE- Continuously Adapting Processing Engine [29]. As shown in Figure 4.1, a CAPE query engine consists of several modules: an Execution Engine, a Local Statistics Gatherer, a Local Adaptation Controller, a Stream Receiver, a Stream Distributor and a Stream Sender. The core of the system is the *Execution Engine* which is in charge of the query plan execution. It schedules operators, controls the statistics collection and calls the Local Adaptation Controller during query processing so that adaptation decisions can be taken to prevent system crash. The Execution Engine uses information obtained from the other modules. The *Statistics Gatherer* calculates and sorts statistics about any part of a query plan, such as operators, queues, and entire plan. We use only light-weight statistics to reduce any statistics overhead. The *Local Adaptation Controller* is in charge of monitoring the load of the system and deciding whether adaptation should start. If the system is overloaded, the Local Adaptation Controller will tell operators to spill data to disk to prevent memory overflow. Vice versa, if incoming data rates are low and the system has ample memory, the Local Adaptation Controller will unspill data previously pushed to disk. Thus the query engine produces complete query results, even

though the results may be delayed depending on the characteristics of the incoming data. The *Stream Receiver, the Stream Distributor and the Stream Sender* are in charge of receiving data from the input streams, placing the newly arriving tuples in the correct queues of the operators and outputting the final results to the end application.

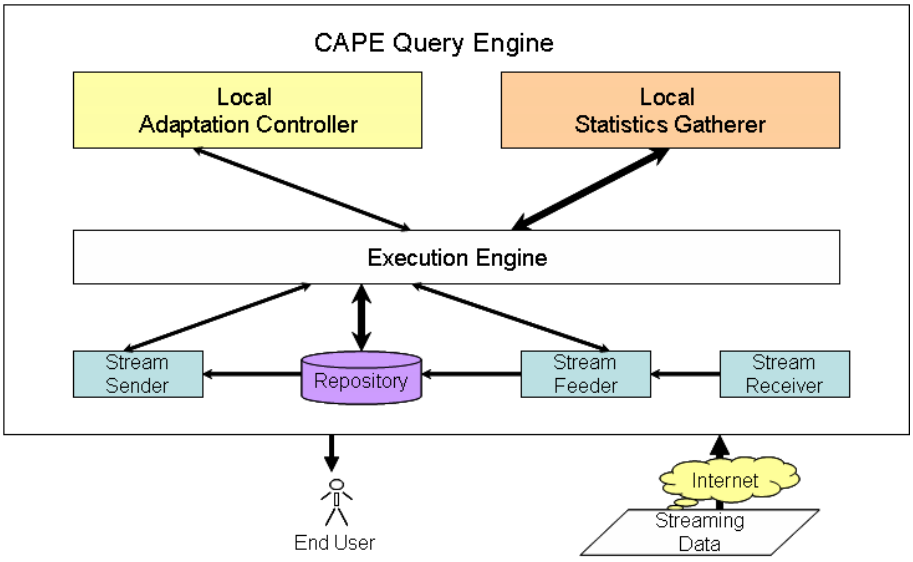

Figure 4.1: System architecture.

To integrate our approach in the existing CAPE framework, we have extended *the Local Adaptation Controller* with two additional sub-components: spilling policies repository and unspilling policies repository. The exact policy classes are configurable parameters in CAPE's initialization file. In theory, any spilling policy can be matched with any unspilling policy, however, we do not recommend this as not all combinations of policies will provide for efficient query plan processing. We have further extended CAPE's query plan xml schema to allow for the definition of windowed operators with different types of window constraints specified on their output.

Chapter 5

Experiments

5.1 Testbed Description

For our experiments we have used two Linux servers with 2 PIII GHz CPUs and 1 G main memory each. Figure 5.1 shows the structure of our testbed. One of the servers was dedicated to query processing. It had installed on it our CAPE query engine. The second server was running both the Stream Generator and the final application.

5.2 Setup and Methodology

Metrics. To compare the performance and effectiveness of the adaptation polices outlined in this thesis, we ran one hour long experiments. The main goal of the adaptation policies is to achieve maximum throughput. We define throughput as the accumulated number of tuples output over time by the query plan. Thus a better policy is a policy achieving higher throughput. Statistical data on throughput as well as other query processing and system load parameters is collected once a minute.

Data sets. Two different types of data sets were used in our experiments. For clarity,

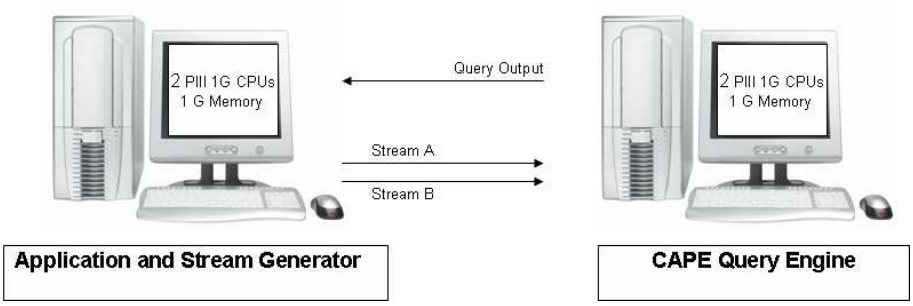

Figure 5.1: Experiment setup.

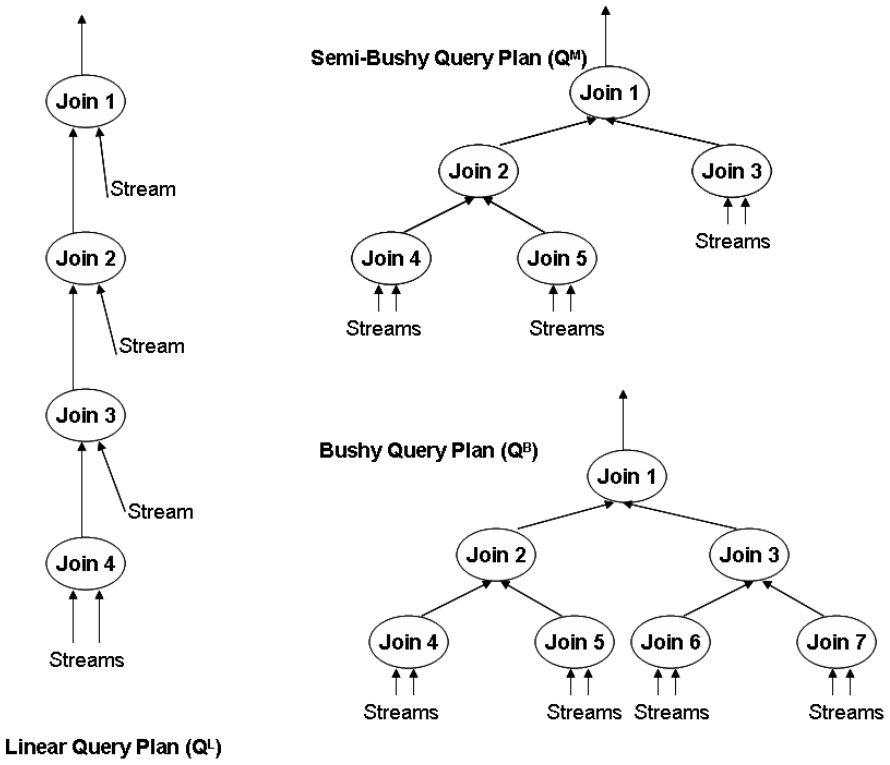

Figure 5.2: Query plans used in the experiments.

Variable name	Notation	Definition
throughput	Θ	Accumulated number of tuples output over time by the query plan.
join ratio	τ	The number of tuples a tuple will be joined with per partitioned window group.

Table 5.1: Definitions table.

we call them D1 and D2. *D1* had approximately the following characteristics per window: one third of the data and thus data partitions had a join ratio of 1, the second third of the data had a join ratio of 2, and the last third of the data had a join ratio of 3. *D2* had approximately the following characteristics: one third of the data and thus data partitions had a join ratio of 1, the second third of the data had a join ratio of 2, and the last third of the data had a join ratio of 4. The definition of join ratio and the notation used to refer to it in the rest of the paper is defined in Table 5.1. The data sets were streamed to the query engine in the form of different data streams at data rates varying from slow to fast tuple arrival using our own stream generator. Tuples were streamed using poisson distribution. The range of data values in each stream varied from 1 to approximately 150,000.

All adaptation policies were tested on three different query plans. The structure of the query plans used in our experiments is shown in Figure 5.2. As can be seen, every query plan is composed of partitioned window join operators only. The data at each operator is partitioned in at least 30 partitions and at most 40 partitions.

Every operator has a window constraint defined on its output. For simplicity, the windows defined on the different operators of the same query plan had uniform characteristics. Unless otherwise stated, they were: a window size of 60,000 ms and a window sliding step of 80,000 ms. We set the memory threshold at the query processor to 200 MB. Thus, whenever the memory consumption by the query processor reaches this limit, the Local Adaptation Controller will initiate data spilling to prevent memory overflow and potential system crash. If memory consumption at the query processor falls to 170 MB or lower, the Local Adaptation Controller will bring previously spilled to disk data back to main memory. Thus available system resources are efficiently utilized. During

data unspilling, only up to 96% of main memory will be filled with tuples. The goal is to prevent the system from being accidentally overloaded by allowing some room for an unexpected spurge of new data. The same principle is applied to network servers whose load is always kept below a certain threshold limit, lower than 100% server utilization.

How much data will be unspilled during an adaptation process will depend on the load of the system at the time the adaptation has been initiated. Unlike unspilling, during a spilling process we spill approximately 30% of all operators' states. If the system's load is in between the spilling and unspilling thresholds for one or more readaptation periods so that neither spilling nor unspilling of data occurs and there are tuples residing on disk, the Local Adaptation Controller will try to unspill as many tuples as possible. This may happen during periods of idle system times when there is available memory. The reasoning is to fully utilize all available free resources.

5.3 Empirical Parameter Tuning

Prior to running experiments assessing the relative performance of the different policies, we ran experiments to determine appropriate values for certain tuning parameters such as the correlation percentage for the Global Synchronized Policy or the readaptation period length for the Local Adaptation Controller. Since these parameters were not the main focus of this thesis, the goal was to find reasonable settings for them and keep them fixed. Below is a discussion of these experiments.

Local Adaptation Controller Readaptation Time. Since data spilling and data unspilling have opposite effects on the memory usage of the query processor, if after each adaptation process the system is not given enough time to adjust there is a risk that the system may start oscillating between spilling and unspilling with each data spilling triggering an unspilling process and vice versa. This would have a negative impact on the

processing of data and ultimately on the final query plan throughput. New data will be processed very slowly, if processed at all. On the other hand, if this system's adjustment time, which we call readaptation time, is too long, a sudden spike in the data arrival may crash the system. Thus, it is important to find a reasonable time range such that the Local Adaptation Controller can correctly evaluate the effectiveness of its decision and take corrective steps in time if this is necessary without jeopardizing the system's performance by waiting too long for its previous decision to take effect.

Figure 5.3 shows the results of our experiments using different readaptation periods. The experiments were conducted using the Linear query plan shown in Figure 5.2 and data set D1 described in Section 5.2. Every experiment was run for one hour. We started the experiments with a data rate of 1 tuple every 30 ms, and then we slowed down the data to 1 tuple every 6000 ms. The tuples' arrival rates were changed every 15 min alternating between the fast and slow speed described above. Thus the system had to operate under fluctuating load conditions allowing us to test both types of adaptation decisions: data spilling and data unspilling.

As it can be seen from Figures 5.3 and 5.4 adapting less often characterized by a longer readaptation period does not necessarily guarantee higher throughput, that is we did not notice a significant trend in the impact of the readaptation time on query plan performance. Within a certain time range, the performance of the query plan seems to be kept at the same level. At the same time a readaptation time of 180,000 ms or more would result in too few adaptation decisions which may not be the best behavior under certain load conditions. A readaptation time of 30,000 ms or less causes too much system oscillation as shown in Figures 5.6 and 5.5. Lets look at Figure 5.6. We can infer how many times the system spilled data to disk by the total number of spilled tuples or unspilled tuples and the slope of the line. During periods of time when no data spilling occurs, the slope of the line is 0, the line is flat parallel to the horizontal axis. Each segment of the

line with slope bigger than 0 indicates a data spilling process has occurred. The same logic applies to the analysis of Figure 5.5. Thus, unless otherwise stated, we used in our experiments a readaptation time of 120,000 ms.

Global Synchronized Policy Correlation Percentage Experiments. As described in Section 3.3 the Global Synchronized Policy aims to provide more efficient memory management by trying to keep partition groups in main memory such that 1) new data is most likely to arrive for these groups and 2) new data has a higher chance of being joined with data already received prior by the operator. To achieve this, the policy uses punctuation pointers to inform operators located downstream the query plan of the timestamps and partition ids of partition groups that have been spilled or unspilled upstream. Details of this process can be found in Sections 2.3, 3.4 and 3.3. Thus, downstream operators can spill or unspill the tuples which would be most likely affected by upstream adaptation actions. Since a tuple can belong to one partition group at one operator and to another one at the next operator, operators need to keep track of the correlation bond which exists between their partition groups and the partition groups of their children. We call this statistics *correlation percentage*. The higher the correlation percentage between two partition groups is, the more tuples partition group one sends downstream to partition group two. Therefore, if we spill partition group one upstream, then partition group two at the next operator will most likely keep receiving less tuples for some period of time after the data spill at operator one. Tuples from partition group one have to start arriving again at operator one and producing join results before more tuples from this partition group can reach operator two.

The Global Synchronized Policies utilize correlation percentage statistics to assert that in their adaptation decisions only partition groups with at least a certain level of correlation percentage are used. For example, if the correlation percentage for the Synchronized Policies is set at 0.1 (10%), then only partition groups with correlation percentage of 0.1

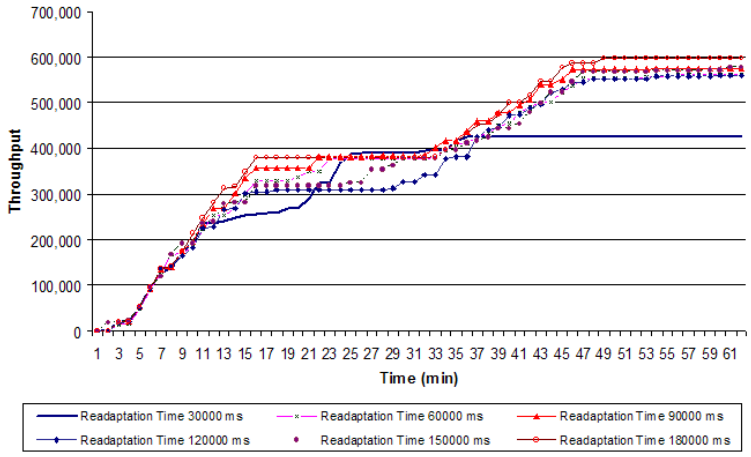

Figure 5.3: Impact of readaptation intervals on query plan throughput- Q^L plan, different readaptation intervals.

Figure 5.4: Impact of readaptation intervals on query plan throughput. Bar graph.

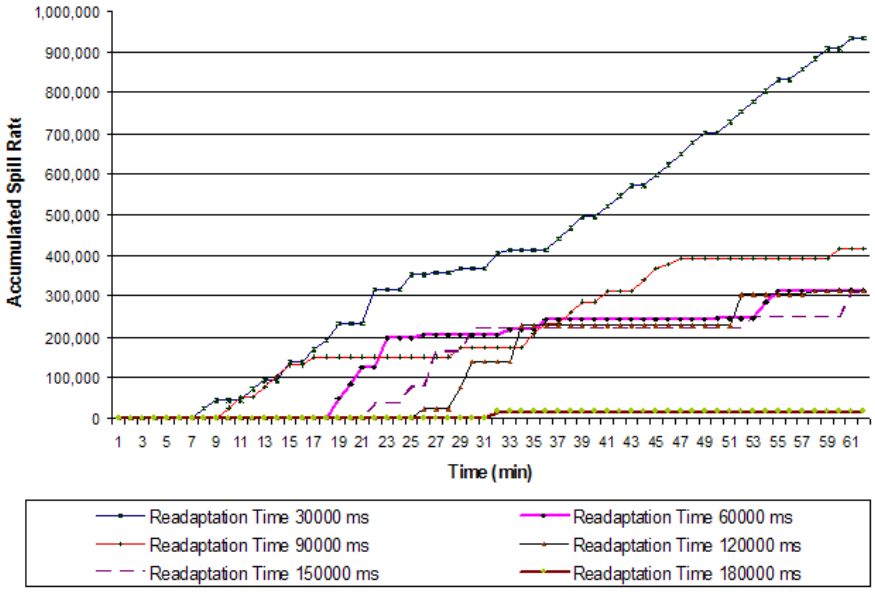

Figure 5.5: Impact of readaptation intervals on unspilling. Accumulated number of unspilled tuples, Q^L plan, different readaptation intervals.

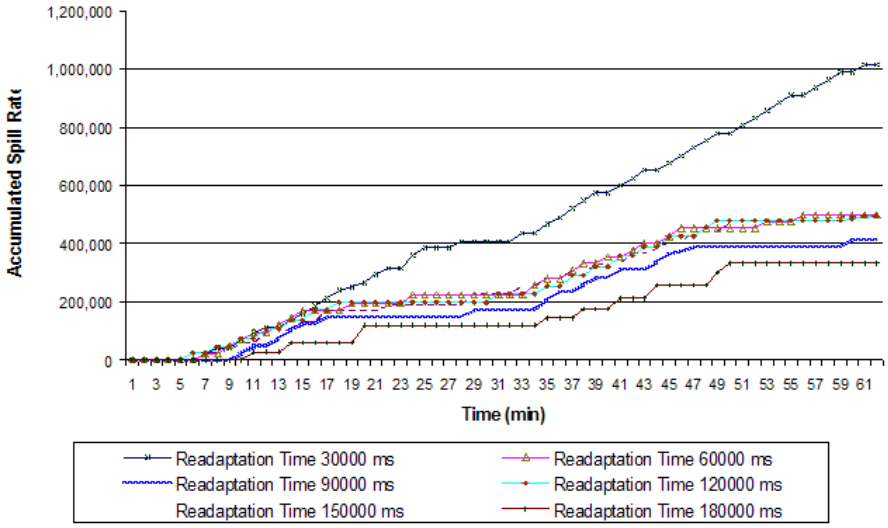

Figure 5.6: Impact of readaptation intervals on spilling. Accumulated number of spilled tuples, Q^L plan, different readaptation intervals.

or higher will be considered for spilling or unspilling by the operators which have received punctuation pointers. Thus smaller values of the correlation percentage entails the spilling and unspilling of more data during query plan execution, and vice versa. A correlation percentage of 1 (100%) means that there is a perfect correlation bond between two partition groups of operators located at two consecutive levels of the query plan. This means that a tuple assigned to partition 1 at operator 1, will always be assigned to partition 1, for example, at the next operator.

In the experiments described below we tried to determine the effect of different correlation percentages on the query plan throughput, invalidation rate, spillling and unspilling. In these experiments we used the Semi Bushy query plan shown in Figure 5.2 and data set D1 as described in Section 5.2. The window characteristics were as described in Section 5.2, namely the window size was set to 60000 ms and the window sliding step was set to 80000 ms.

To reduce the overhead of collecting correlation percentage statistics, statistics were collected by each operator with 10% probability. This means that an operator would update its correlation statistics data structure only for one out of ten tuples. The statistical data for the Semi Bushy Query Plan experiments are provided in Table 5.2. The table displays summarized information across all partition groups. We calculate correlation percentage between two partition groups by counting first how many tuples with partition ID *n* at a child operator map to partition group *k* at a parent operator and second the total number of tuples with partition ID = n received at the parent operator. Then we divide the first counter over the second counter. The ”Operator Id” column of Table 5.2 stores id of the operator which has collected correlation percentage statistics. The ”Correlation Percentage” column contains a particular correlation percentage value. The ”Frequency as Percentage” column shows what percentage of the partition groups at that operator have a correlation percentage with the value specified at the same row in the

”Correlation Percentage” column. As the table shows, operators connected to streams only have a correlation percentage of 1 across all partitions because 1) tuples received by a leaf operator will be always joined on the same column and 2) we do not repartition data dynamically.

Table 5.2 shows that approximately 30% of the partition groups have a correlation percentage of 0.02 or less, more than 70% of the partition groups have a correlation percentage of 0.05 or less. As the table shows very few partition groups, only about 10% of all sampled partitions have a correlation percentage of 0.1 or more. In the experiments we varied the correlation percentage values from 0.02 to 0.25. All values are provided in Table 5.2.

Figures 5.7, 5.8, 5.9 and 5.10 show the results of these experiments. As expected, higher values of correlation percentage cause less spilling and unspilling than lower values. Thus due to the overhead of the increased number of disk accesses, the experiments during which the correlation percentage was set to 0.02 and 0.05 show lower throughput. At the same time there is not a clear trend that increasing the correlation percentage values for the Synchronized Policies guarantees higher throughput. As can be seen in Figure 5.7, highest throughput was achieved when the correlation percentage was set to 0.07. Throughput was 1,044,382 tuples for one hour as opposed to 994,263 tuples when the correlation percentage was set to 0.25. As Figure 5.10 shows, higher levels of correlation percentage account for more tuples being invalidated as tuples are moved faster through the query plan. Since experimental results show that the correlation percentage statistics collected by our operators has on average the values shown in Table 5.2, unless otherwise stated in our experiments we have used an average value of 10% for the Global Synchronized Experiments.

Linear Query Plan Dequeue Ratio. We define as dequeue ratio the number of tuples an operator dequeues from a queue connected to another operator, before dequeuing a

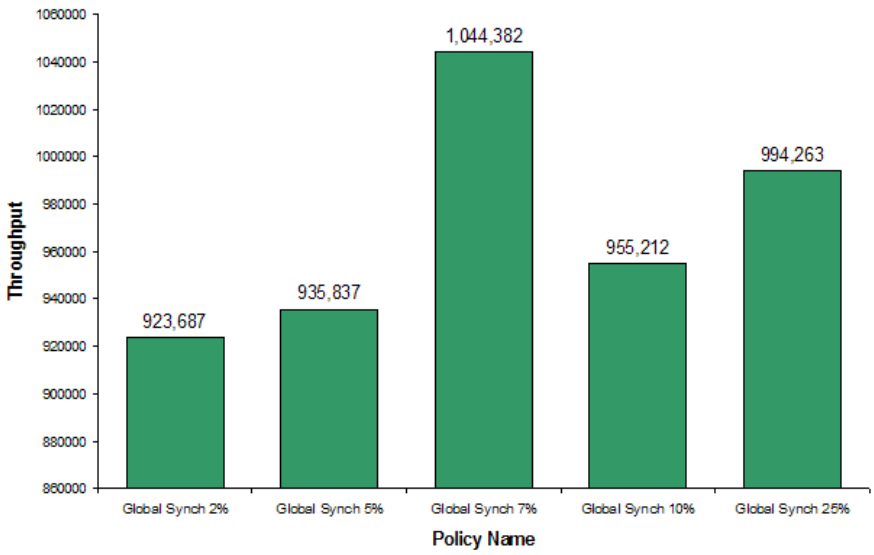

Figure 5.7: Impact of correlation percentage on throughput. Global Synch Policy, Q^M plan.

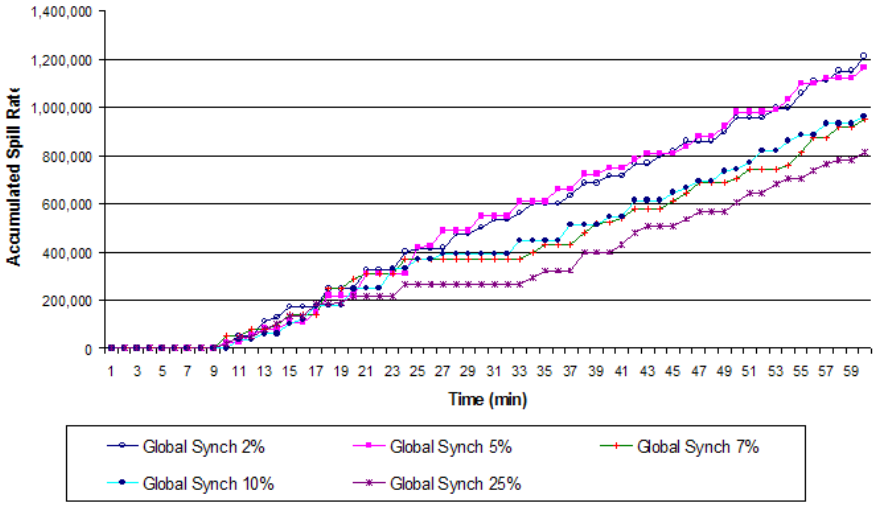

Figure 5.8: Impact of correlation percentage on spilling. Accumulated number of spilled tuples, Global Synch Policy, Q^M plan.

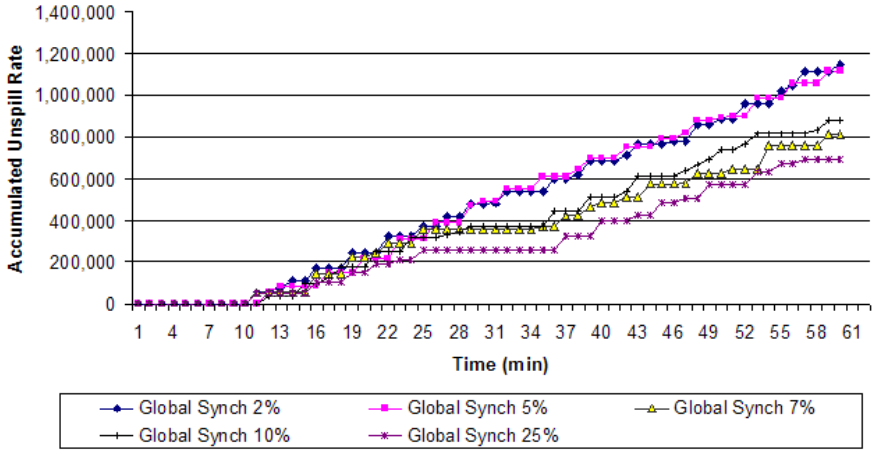

Figure 5.9: Impact of correlation percentage on unspilling. Accumulated number of unspilled tuples, Global Synch Policy, Q^M plan.

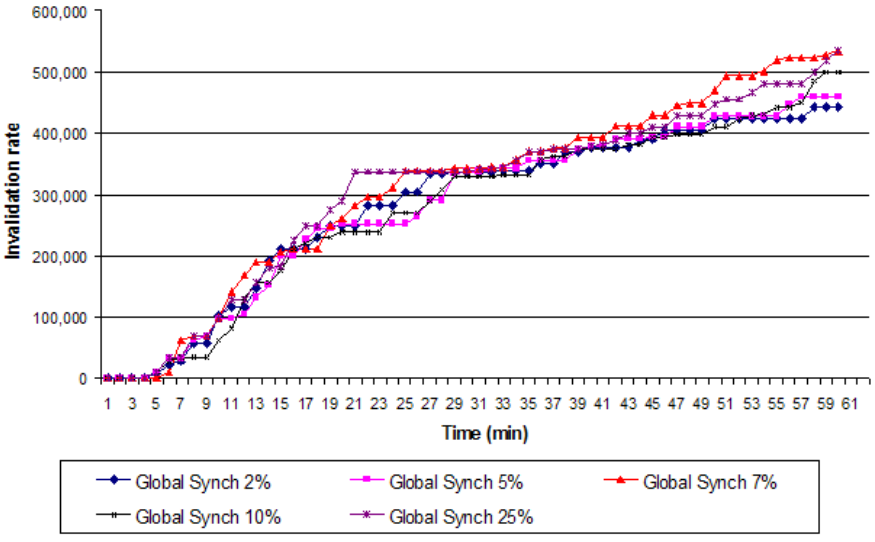

Figure 5.10: Impact of correlation percentage on invalidation. Invalidation rate, Global Synch Policy, Q^M plan.

Operator Id	Correlation Percentage	Frequency as Percentage
Operator 1	0.02	0.33
	0.05	0.83
	0.07	0.89
	0.10	0.91
	0.25	0.95
	0.40	0.97
	1	1
	Mode	0.03
Operator 2	0.02	0.38
	0.05	0.74
	0.07	0.86
	0.10	0.92
	0.25	1
	0.40	1
	1	1
	Mode	0.01
Operators 3,4 and 5 are connected directly to streams. Thus all partition groups have a correlation percentage of 1.		

Table 5.2: Q^M plan, correlation percentage statistics.

tuple from a queue connected directly to a stream. For example, if operator 1 has two input queues. Input queue one is connected to Operator 2 and input queue 2 is connected to Stream A. A dequeue ratio of five means that Operator 1 will dequeue 5 tuples from queue 1 before dequeuing a tuple from queue 2. By assigning a higher weight to operator propagated tuples we try to move punctuation pointers faster through the query plan. This is achievable since punctuation pointers are actually tuples interleaved in the stream of output results of an operator. A skewed dequeue ratio will have an impact only on query plans which are composed mostly of operators with mixed types of input queues, that is at least one input queue of such operators comes from another operator and at least one input queue comes directly from a stream.

In the next set of experiments we investigated the effect of faster punctuation pointers propagation on the query plan throughput of the different policies. The experiments were conducted using the Linear Query Plan and data set D1. Data rates were changed every 15 min from fast to slow and vice versa. By fast data rate we define a tuple sent every 30 ms, a slow data rate is a tuple sent every 6000 ms. The results of the experiments are shown in Figure 5.11. We did not see a significant trend in the impact of different dequeue ratios on the throughput of different policies. This may be the case as after one hour we stop

the experiment leaving any data on disk and in the queues unprocessed, thus to a certain extent the subsets of the input data stream set processed by the query plan varies.

5.4 Comparative Evaluation of the Different Adaptation Policies

In these sets of experiments we studied how the different adaptation policies affect query plan throughput and memory management. Every set of experiments was done by keeping most parameters such as readaptation time, query plan type, data rates and data sets fixed. Across experiments from the same set we changed only the adaptation polices. Across different experiment sets we changed the query plans and the data sets used, as well as the data rates. Unless otherwise stated, we kept the readaptation intyerval fixed at 120000 ms.

We studied four different adaptation policies since each spilling policy was paired up with its counterpart unspilling policy to form one adaptation policy. The four polices are: Random Adaptation Policy, Local Adaptation Policy, Global Unsynchronized Adaptation Policy and Global Synchronized Adaptation Policy. For each set of experiments we ran an experiment without imposing any memory constraints on the query processor. The goal was to see the optimal query plan throughput for the given set of parameters: query plan, data set and data rates. Experimental results shown in Figures 5.12 and 5.7 indicate that Global Adaptation Policies consistently achieve higher query plan throughput than the others. On average the Global Synch Policy performed 3% to 4% better than the Global Unsynch Policy, around 8% better than the Local Policy in some case more than 20% better than the Random Policy. The Random Policy scored worst not only on query plan throughput but also on data invalidation.

Figure 5.12 shows a summary of the results for some of the experiments we have

conducted. Other experiments show similar results. As the figure shows, Global Policies perform consistently better than the Local and the Random Policies. The figure also shows that Global Synch Policy consistently performs better than the Global Unsynch Policy.

Figures 5.13, 5.14, 5.15 and 5.16 show the throughput, invalidation rate, the spill and the unspill rate of a set of experiments using the Bushy Query Plan. In this set of experiments the readaptation interval was set to 120000 ms and the correlation percentage for the Global Synch Policy was set at 10%. Data rates were changing every 15 minutes from fast to slow. The fast data rate was set to 1 tuple every 45 ms and the slow data rate was set to 1 tuple every 6000 ms. The data had an approximate join ratio per window of 1 for one third of the partitions, 2 for the second third of the partitions and 3 for the last third of the partition groups. The window characteristics of window per operator were: window size of 60000 ms and window sliding step of 80000. As Figure 5.13 shows the Global Synch Policy performed 2% better than the Global Unsynch Policy. The overhead of the Global Synch Policy in comparison to other policies does not seem to be much. Even though as Figure 5.15 indicates that all policies spilled equivalent amounts of data, the Global Synch Policy output was approximately 10000 tuples more than that of the Global Unsynch Policy and 30000 tuples more than that of the Local Policy, which is equivalent to a 5% improvement. The Random Policy produced 14% less tuples than the Global Synch Policy. The Random Policy also performed worst in terms of tuple invalidation as shown in Figure 5.14 .

Figures 5.17, 5.18, 5.19 and 5.20 show the throughput, invalidation rate, the spill and the unspill rate of another set of experiments using. In this set of experiments we used the Linear Query Plan. The readaptation time was set to 120000 ms and the correlation percentage for the Global Synch Policy was set at 10%. Data rates were changing every 15 minutes from fast to slow. The fast data rate was set to 1 tuple every 45 ms and the slow data rate was set to 1 tuple every 6000 ms. The data had an approximate join ratio

Figure 5.11: Impact of dequeue ratios on throughput. Different dequeue ratios, Q^L.

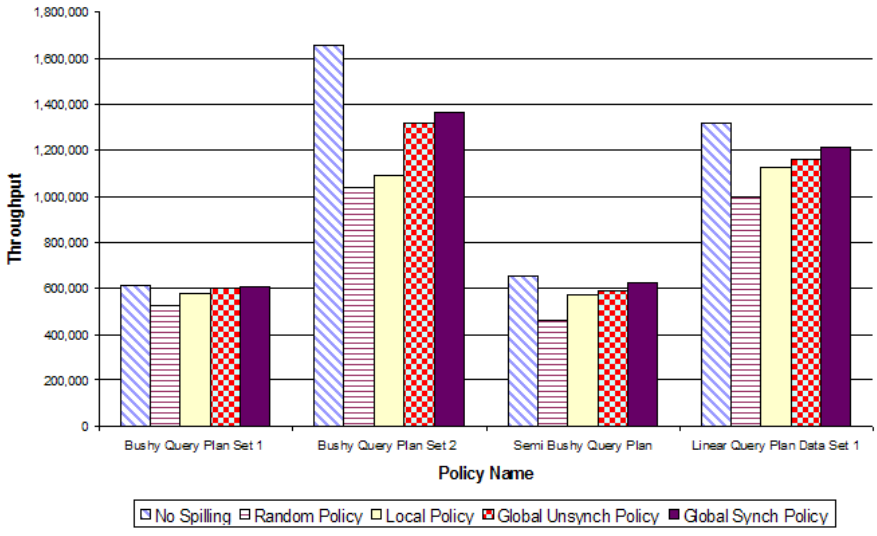

Figure 5.12: Comparative evaluation of throughput across all adaptation policies.

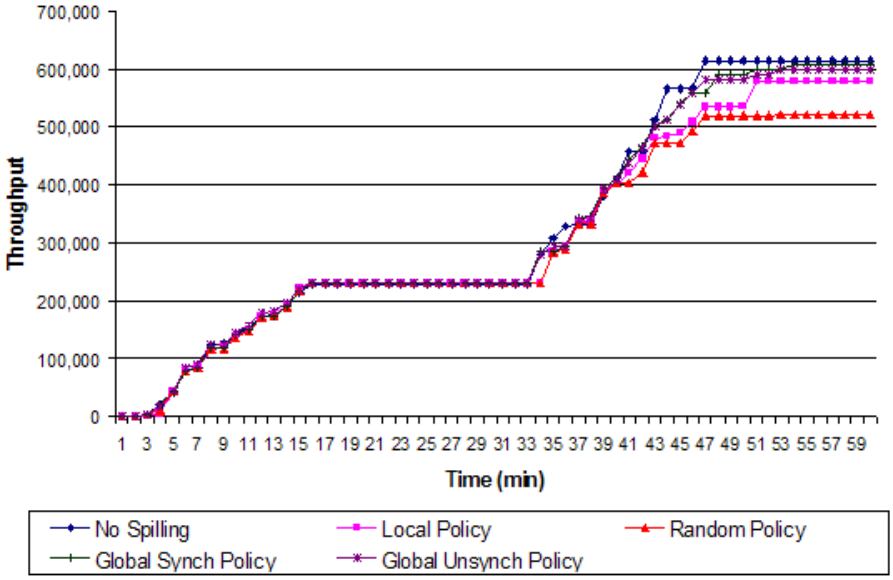

Figure 5.13: Throughput, Q^B plan, data rate 45/6000. All adaptation policies.

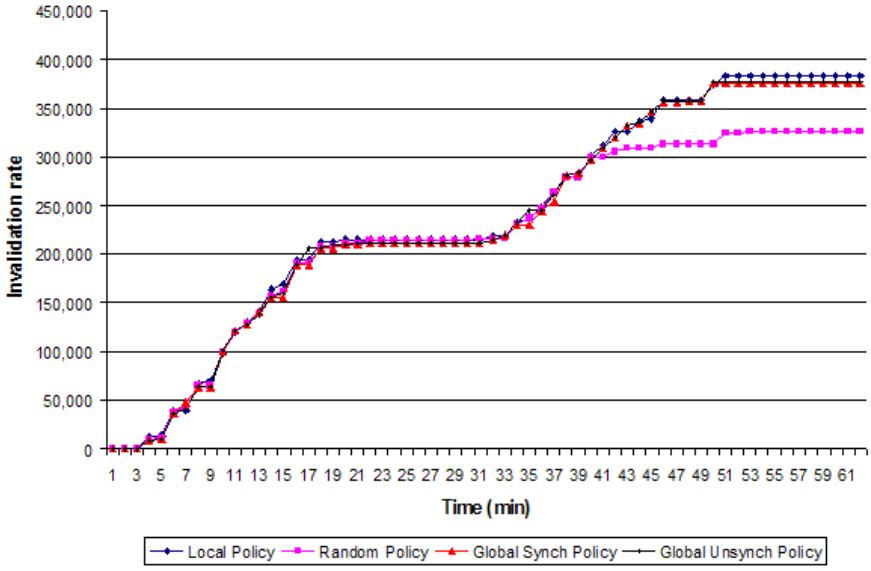

Figure 5.14: Invalidation rate, Q^B plan, data rate 45/6000. All adaptation policies.

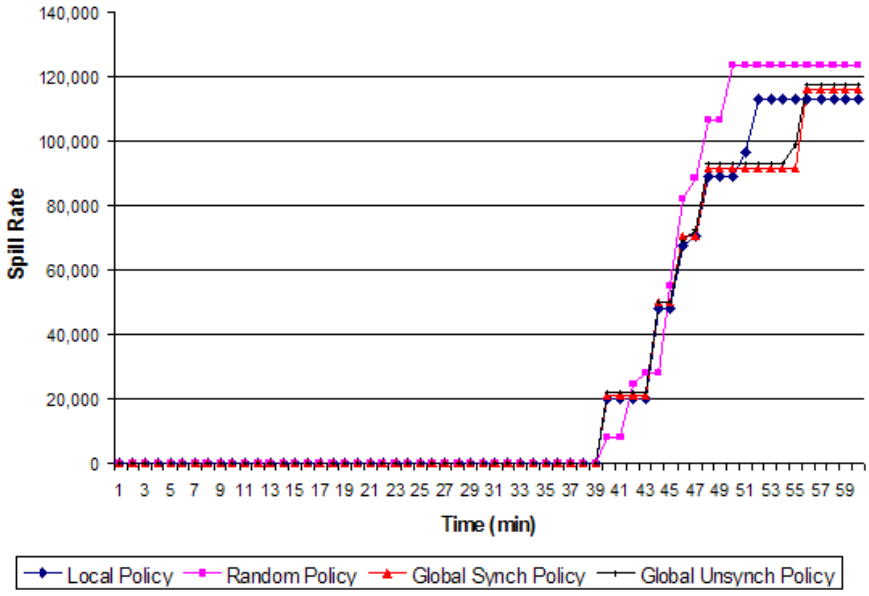

Figure 5.15: Spill rate, Q^B plan, data rate 45/6000. All adaptation policies.

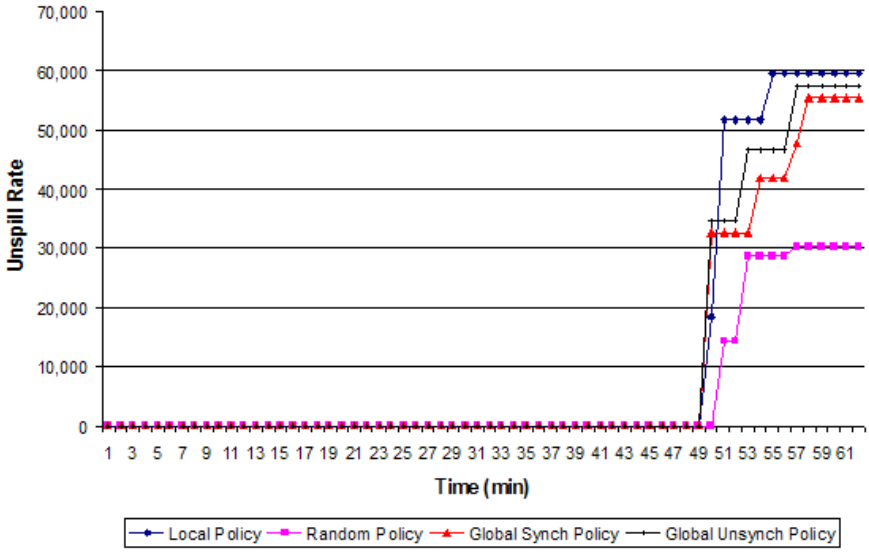

Figure 5.16: Unspill rate, Q^B plan, data rate 45/6000. All adaptation policies.

per window of 1 for one third of the partitions, 2 for the second third of the partitions and 4 for the last third of the partition groups. The window characteristics of window per operator were changed: window size was set to 40000 ms and the window sliding step was set to 90000. As Figure 5.17 shows the Global Synch Policy performed 4% better than the Global Unsynch Policy by outputting 50000 tuples more. Even though as Figure 5.19 all policies spilled equivalent amounts of data, the Global Synch Policy performed 8% better than the Local Policy and 18% better than the Random Policy which was equivalent to more than 200000 more throughput. As Figure 5.20, even though the Local Spill Policy unspilled more data during the query plan execution it still produced less output tuples. Figure 5.18 shows that the Random Policy still had the worst invalidation rate in comparison to the other policies. The Global Synch Policy managed to invalidate most tuples.

Results from additional experiments are included in Appendix A of the thesis.

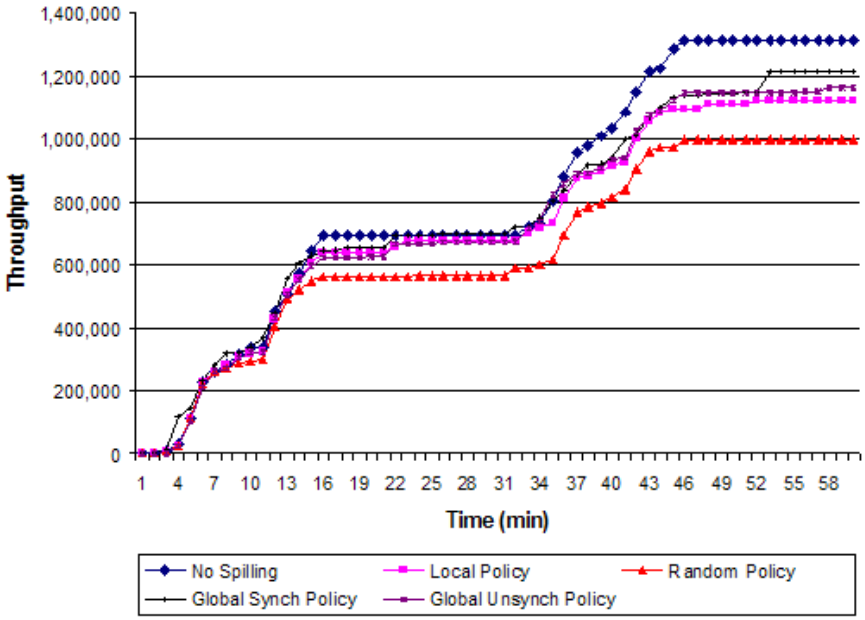

Figure 5.17: Throughput, Q^L plan, data set D2. All adaptation policies.

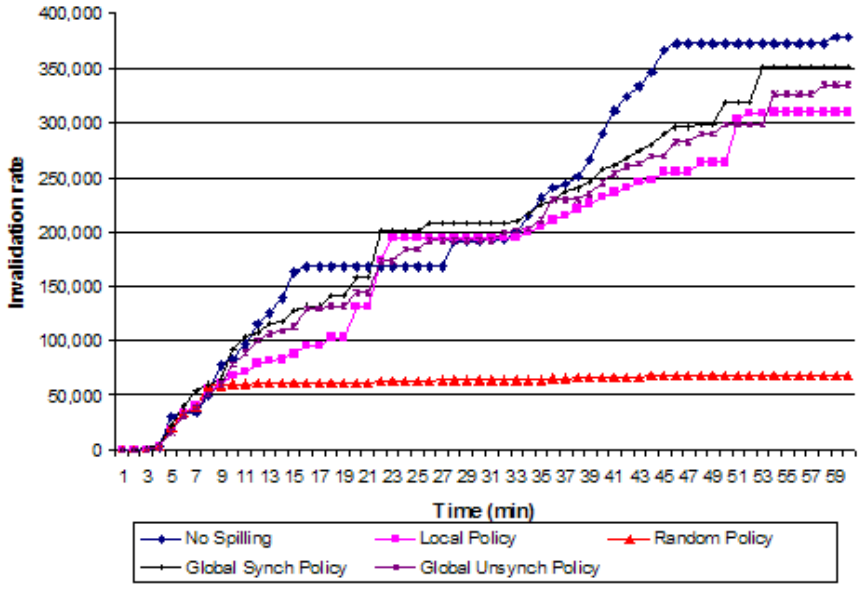

Figure 5.18: Invalidation rate, Q^L plan, data set D2. All adaptation policies.

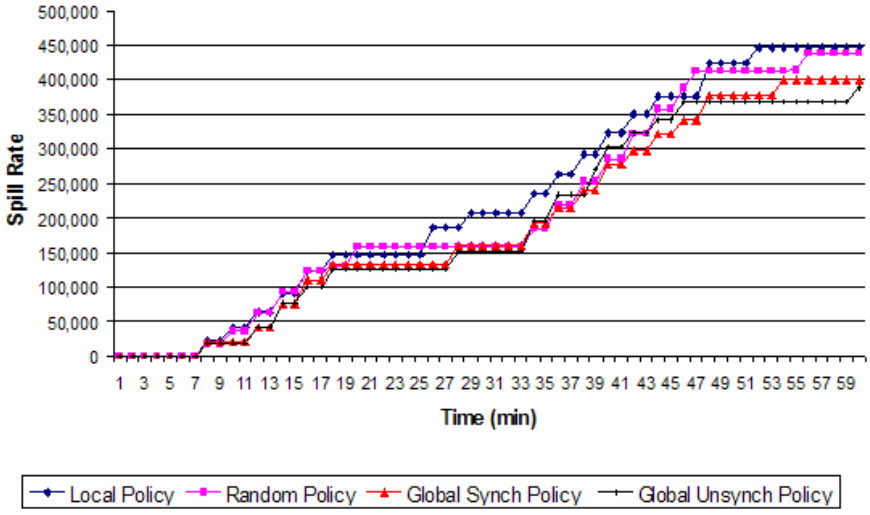

Figure 5.19: Spill rate, Q^L plan, data set D2. All adaptation policies.

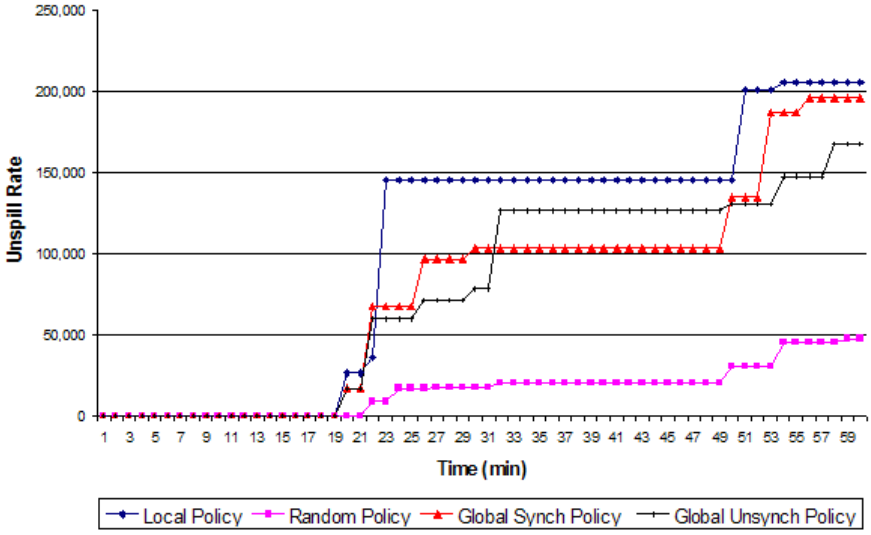

Figure 5.20: Unspill rate, Q^L plan, data set D2. All adaptation policies

Chapter 6

Related Work

This thesis is based on research on the processing of continuous queries and data partitioning in the context of streaming environment. The systems designed to process continuous queries are called data stream management systems. As is known, data stream management systems operate under requirements and constraints different than those imposed on traditional database systems. Unlike traditional database systems, data stream management systems do not deal with finite amount of data, do not answer queries on data that has been already stored on disk and possibly analyzed, and do not return as query results exact, finite sets of tuples. The workload a data stream management system has to handle depends not only on the type and number of queries in has to process but also on the arrival rates of the incoming data which can be quite unpredictable. Since DSMS and traditional database systems have to operate under different conditions, adaptation methods developed for traditional database systems may no longer be applicable to the streaming environment. Currently, optimization methods for DSMS include at the operator level exploiting an operator's selectivity [3, 29], using operator's punctuations for state purging [5, 29]; at the scheduler level various operators' scheduling techniques [29, 22, 4], and query approximation methods such as load shedding [35]; at the query plan level the dis-

tribution of the query plans across multiple machines [29], dynamic query plan migration [36] and operators reallocation [9, 27].

Efficiently handling critical resources such as main memory is a major concern in the design of DSMSs. One way of reducing system load is distributing query processing over multiple machines. Since the DSMS research field has been developing rapidly for the past few years many centralized and distributed query processing prototype engines such as Stream [30], Telegraph [22], Aurora [2, 14], CAPE [36], D-CAPE [28, 21], Aurora* and Medusa [35], Borealis [1, 12, 34] and others using different query optimization techniques have been developed.

STanford stREam datA Manager (STREAM) [30] is a general-purpose system for processing continuous queries over multiple continuous data streams and stored relations, which has been designed to handle high-volume and bursty data streams. Telegraph [22] is a continuously adaptive, continuous query system based on the eddy query processing framework which uses crossquery sharing of system resources such as computation and storage. Telegraph uses very fine-grained tuple-level adaptation techniques which allows tuples to be dynamically rerouted through operators based on recent operators' cost and selectivity statistics. CAPE [36] is a general-purpose DSMS with a heterogeneous-grained adaptivity that exploits dynamic metadata at all levels in continuous query processing, including the query operator execution, memory allocation, operator scheduling, query plan structuring. Our operator and adaptation policies have been incorporated and tested using the CAPE framework. D-CAPE [28] is the distributed version of CAPE. It has a central distributed architecture with a dedicated distribution manager managing a set of query processors installed on a local cluster of machines connected with a high speed network. Unlike D-CAPE, Aurora*/Medusa [35, 2] focus on research issues related to the processing of continuous queries over a large network. Aurora* is designed as a distributed system without a centralized controller to monitor its performance. In Aurora*

every processor communicates with its neighbors. During periods of high load a processor tries to offload some of its processing tasks to a less loaded neighbor [13, 35]. Medusa describes an agoric model [7] for the cooperation of different distributed DSMS operating under different administrative domains [35]. Borealis [12] is the successor of the Aurora* and Medusa projects aimed at incorporating features such as failure-detection and failure recovery [26], dynamic revision of query results [12], dynamic query modification and others [12] and QoS-based optimization [12]. Like Medusa, the Borealis system can operate in federated mode stretching over different administrative domains.

Queries supported by our framework can be written in CQL [25]. CQL is an SQL-based continuous query language for the expression of general-purpose continuous queries over streams and updatable relations. It incorporates window semantics. CQL authors aim at exploiting well researched relational semantics principles. [25] discusses the semantics of CQL, presents a comparison analysis of CQL to other continuous like query languages and presents the implementation of CQL in STREAM. [25] also presents the idea of *heartbeats*. Heartbeats are an additional meta-input to the system. They represent timestamps with the semantics that after the arrival of a heartbeat τ the system will receive no more tuples with timestamp of τ or lower [25]. Heartbeats can be generated in different ways. They are needed as CQL semantics "assumes a discrete, ordered time domain T" [25] and network transmission over remote sources has unpredictable character which does not guarantee in-order data transmission. Heartbeats are discussed in more details in [33]. In our work we assume that tuples arrive in order from the data sources either using source synchronization or heartbeats- the mechanism which ensures in-order tuple arrival is irrelevant to us as it is outside the scope of this thesis. The Partitioned Window Operator we have designed can handle out of order tuple arrival using punctuation messages.

CQL is based on "two data types: streams and relations, and on three classes of opera-

tors over these types: operators that produce a relation from a stream (stream-to-relation), operators that produce a relation from other relations (relation-to-relation), and operators that produce a stream from a relation (stream-to-relation)". [25] defines a stream S "as a (possibly infinite) bag (multiset) of elements $< s, \tau >$, where s is a tuple belonging to the schema of S and $\tau \epsilon T$ is the timestamp of the element", and a relation (R) as: "a mapping from T to a finite but unbounded bag of tuples belonging to the schema of R".

Unlike CQL which assumes slide-by-tuple semantics and uses a predefined timestamp or tuple order number, [18] introduces new user defined attributes: SLIDE and WATTR, which can be used to express windows with different sliding steps based on different tuple attributes.

[18] propose a framework for defining window semantics which can be used to express many window types and a framework which uses their window semantics to evaluate different types of window aggregate queries. The advantages of their approach is that each tuple is processed only once as it arrives and at most only one copy of each tuple is stored if the tuple needs to be buffered. This is similar to our implementation. We also process each tuple on the fly and we store only one copy of it, no matter how many windows a tuple belongs to. Unlike [18] who focus on stateless aggregate operators such as min and max, in this work we focus on the implementation of a stateful window join operator. In our work we use a tuple's timestamp as the window attribute, however, this can be easily changed and other models can be plugged in. As discussed by [18] disorder in tuples' arrival can be handled by using punctuations.

[17] further discusses the implementation of window aggregate operators using the window semantics described in [18].

The design of our Partitioned Window Join Operator is based on prior research on the implementation of join operators optimized for the streaming environment such as Flux [27] and Eddy [3]. We have also looked at the implementation of join operators which

use data spilling and other techniques to handle bursty dataflows such as XJoin [32].

Flux [27], Fault-tolerant Load-balancing eXchange, is a dataflow operator that encapsulates adaptive state partitioning and dataflow routing. Flux is inserted between a partitioned producer-consumer pair in a parallel dataflow pipeline. [27] provides adaptive repartitioning while the pipeline is still executing. Flux uses a buffering and reordering mechanism to handle short-term load imbalances as well as a mechanism for detecting across cluster imbalances and online repartitioning of state accumulated in lookup-based operators. Unlike [27], our Partitioned Windowed Join Operator partitions the data at the beginning of the query plan execution and this partitioning is kept static throughout the query plan processing. Similar to Flux, however, we use multiple mini-partitions to reduce the overhead of our state-spilling adaptation technique.

Eddy [3] is a query processing mechanism which allows for a tuple level query plan adaptation. As eddy encapsulates the scheduling of its participating operators, tuples entering the eddy can flow through operators in a variety of orders allowing for a continuous run-time operator reordering. Each operator participating in the eddy has one or two inputs that are fed tuples by the eddy, and an output stream that returns tuples to the eddy. Each tuple entering an eddy carries its own execution history implemented as bitmaps of ready and done bits which encode what operators a tuple has been already processed by and what operators the tuple has to be sent to. An eddy routes each tuple to the next operator based on the tuples execution history and statistics maintained by eddy [31]. [31] focuses on the design, implementation, and performance of a distributed eddies in which operators themselves decide on where next a tuple should go based on the execution history of the tuple and statistics maintained at the operator. [31] discusses different routing policies. Despite allowing for the implementation of very flexible tuple-level adaptation polices, Eddies seem to incur too much memory and computation overhead per tuple. Our adaptation approach is completely different than Eddy as it is at the state level of

operators. We do not do dynamic rerouting of tuples.

XJoin [32] is a non-blocking join operator optimized to produce initial results quickly and to hide intermittent delays in data arrival by reactively scheduling background processing. XJoin is based on symmetric hash join. It is designed to handle data access to traditional database systems over wide-area networks. XJoin proceeds in three stages. Stage 1 proceeds as long as both inputs to the operator keep sending data. Stage 2 is activated when when the first stage is timed out on both of its inputs. Based on different cost estimates, then data stored on disk is brought to main memory and processed using memory resident data from the other queue. Stage 3 is the final clean-up stage which is activated when all data has been received from both sources. XJoin is designed to work over traditional database tables accessed across distributed networks.

Similar to XJoin [32] and progressive merge join, the hash-merge join algorithm presented in [23] focuses on processing join results over finite datasets accessed across remote networks. The algorithm, which is based on a two-way join operator, aims at outputting join results as early as possible. [23] consists of two phases a hashing and a merging phase. During the hashing phase incoming tuples are stored in in-memory hash buckets and join results are produced from these in-memory tuples. When the memory becomes full, hash buckets are sorted and flushed to disk. Disk residing tuples are joined during the second merging phase of the algorithm. Unlike [32] which pushes to disk the largest partition of one of the data sources only, [23] uses an Adaptive Flushing Policy which simultaneously pushes to disk hash buckets from both sources. The Adaptive Flushing Policy aims at keeping the memory balanced between the two remote sources, that is the ratio of tuples from the two data sources residing in memory has to be within some predefined limits. [23] discusses the advantages and disadvantages of several different flushing policies such as flush-all policy, flush largest partition first,and flush smallest partition first policy. While the main drawback of the flush smallest partition first policy

is the greater number of I/Os incurred during the merging phase since a bigger number of small data blocks have to be read from disk, the flush largest partition first policy increases the amount of time it takes to output join results during the hashing phase after a flush–to-disk operation has taken place since there are less tuples in main memory which reduces the probability of a join result when a new tuple arrives. The same line of reasoning can be applied in the analysis of the impact of spilling to disk policies applied to the context of continuous query processing.

Accuracy of the received query results is not always as vital to some final applications as is the requirement of receiving constant results. When serving such applications, during periods when incoming data exceeds the capacity of the DSMS to process it, a DSMS may decide to discard a certain portion of its load to prevent system crash. As defined in [24], the process of dropping excess load is called data shedding. [24] implements data-shedding as drop operators dynamically inserted or removed into different levels of the query plan. The paper discusses two types of drop operators: random drops which simply drop a dynamically calculated portion of the tuples, and a semantic drop which discards tuples with the lowest utility. Our semantic policy uses the same idea of assigning different levels of importance to tuples based on their values. However, instead of permanently discarding the tuples with lowest utility we simply delay their processing by temporarily spilling them to disk. [24] presents also the idea of QoS (quality of service) associated with each application served by the DSMS. In [24] is modeled as a ”set of functions that relate a parameter of the output to its utility”. In Aurora, in which this framework has been implemented, QoS is expressed in three functions: ”a latency graph, a value-based graph, and a loss-tolerance graph” [24].

Similar to [19] we use punctuations to optimize the processing of the partitioned window operator presented in this thesis. However, the punctuation messages which we use have a broader meaning. [19] defines punctuation as ”an ordered set of patterns, each

corresponding to an attribute of the tuple". We overload the punctuation message with extra information about the stage of processing which has been just completed by the sending operator. We use punctuation pointers not only to invalidate no longer needed tuples from an operator's state but also to spill or unspill partitioned windows across operators in a synchronized way. Unlike [19] which focuses on the processing of continuous sliding windows, we focus on the processing of hopping windows. [19] presents join optimization techniques for the processing of data "with two types of constraints imposed on it: time-based constraints (sliding windows) and value-based constraints (punctuations)". The partitioned window join operator proposed in this thesis focuses on the processing of data with time-based constraints only imposed on it.

Dynamic plan migration is another technique for optimizing the processing of continuous queries. Unlike the techniques proposed in this thesis which work at an operator level, dynamic plan migration is applied at query plan level. [36] defines dynamic plan migration as "the on-the-fly transition from one continuous query plan to a semantically equivalent yet more efficient plan". [36] presents two alternative strategies, called the moving state strategy and the parallel track strategy, and cost models to analytically compare them. The parallel track strategy continuously keeps outputting results even while plan migration takes place. The moving state strategy, on the other hand pauses plan execution during the migration phase.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Query plans with state-intensive operators may consume a lot of system resources when faced with spikes in the arrival patterns of new data. Thus a data stream management system needs efficient adaptation policies with minimal overhead so that all data can be processed and none dropped even during periods of high system load. The requirement for complete and accurate query results is presented by many applications such as financial analysis systems, mobile hospital applications and etc. Efficient adaptation policies can be designed by utilizing the different dependencies which exist among operators in a query plan.

In this work we have identified two types of such dependencies, namely content- and time-based dependencies. We have also done the following tasks:

1. A new partitioned window join operator with the ability to spill and unspill data to disk on demand has been designed.

2. We further extend the semantics of punctuations embedded in the data stream to encode information of the processing stages completed by an operator. Such infor-

mation is used for the correct processing of out-of-order tuples and for the design of efficient data invalidation policies.

3. A new adaptation policy to synchronize the work of operators in a query plan has been designed. The policy uses metadata about the stages of query execution propagated down the query plan tree by operators and partition level statistical data to make better memory management adaptation decisions.

4. We have designed several different adaptation policies with different levels of query plan synchronization.

5. All the policies have been implemented and integrated into a data stream management system called CAPE.

6. Experiments on the relative performance of the different adaptation policies have been carried out using a real software system, not simulation.

Our experiments prove that despite the higher overhead of a more synchronized adaptation approach, our consolidate strategy provides for better query plan performance and higher plan throughput during periods of continuous bursts of high data rates. Such an integrated policy proves to be more efficient at memory management based on invalidation rates than a Random Adaptation Policy with very little computational overhead.

7.2 Future Work

Future work may include testing different cost models which can be easily plugged in our framework. Experiments are necessary on the Semantic Adaptation Policy which has not been tested yet. The experiments in this thesis cover only query plans with hopping windows imposed on their output. It will be interesting to see what the performance of the

adaptation policies described in this work will be on query plans with continuous windows imposed on them. New ”processor overload” strategies as well as new adaptation policies and other operators besides joins can be also developed and plugged into our framework.

An interesting future task to do will be the integration of our adaptation techniques on a distributed stream management system. In a distributed environment work can be kept evenly distributed among processors. During spikes in the arrival of data an overloaded processor's work can be offloaded to another less loaded query processor. In such an environment, a query processor can either spill data or offload work to a different query processor. How does a processor decide what adaptation technique to apply? Such and other problems have been already discussed in [9]. It will be interesting to combine both [9]'s work and this thesis.

Bibliography

[1] D. Abadi, Y. Ahmad, and et. al. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] D. Abadi, D. Carney, and etl. Aurora: a new model and architecture for for data stream management. In *VLDB Journal*, pages 120–139, 2003.

[3] R. Avnur and J. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[4] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: operator scheduling for memory minimization in data stream systems. In *SIGMOD*, pages 253–264, 2003.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[6] S. Babu, L. Subramanian, and J. Widom. A data stream management system for network traffic management. In *NRDM*, 2001.

[7] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *NSDI*, pages 197–210, 2004.

[8] B.Babcock, R. S.Babu, and M.Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM SIGMOD*, pages 253–264, 2003.

[9] B.Liu. Scalable integration view computation and maintenance with parallel, adaptive and grouping techniques. In *Ph.d. Dissertation*, 2005.

[10] B.Liu and E. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, pages 829–840, 2005.

[11] B.Liu, Y.Zhu, and E. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *SIGMOD*, pages 347–358, 2006.

[12] Borealis Team. The design of the borealis stream processing engine. In *Technical Report*, October 3, 2004.

[13] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[14] M. D. Abadi, U. Cetintemel and etc. Aurora: a data stream management system. In *SIGMOD*, pages 666–666, 2003.

[15] D.J.Dewitt, J.F.Naughton, D.A.Schneider, and S.Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[16] J.Kang, J. Naughton, and S.Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[17] J.Li, D. Maier, K.Tufte, V.Papadimos, and P. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[18] J.Li, D. Maier, K.Tufte, V.Papadimos, and P. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD*, pages 311–322, 2005.

[19] L.Ding and E. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107, 2004.

[20] B. Liu, M. Jbantova, and E. A. Rundensteiner. Optimizing state-intensive non-bocking queries using run-time adaptation. In *SSPS07, Workshop co-located with ICDE*, 2007.

[21] B. Liu, Y. Zhu, and et. al. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB Demo*, pages 1338–1341, 2005.

[22] S. Madden, M. Shah, and J. Hellerstein. Continuously adaptive queries over streams. In *ACM SIGMOD*, pages 49–60, 2002.

[23] M.Mokbel, M.Lu, and W.Aref. Hash-merge join: a non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–263, 2004.

[24] N.Tatbul, U. Cetintemel, S. Zdonik, M.Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[25] S.Babu and J.Widom. The cql continuous query language: semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[26] M. Shah, J. Hellerstein, and E. Brewer. Highly available, fault tolerant dataflows. In *SIGMOD*, pages 827–838, 2004.

[27] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Data Engineering*, pages 25–36, 2003.

[28] T. Sutherland, B. Liu, M. Jbantova, and E. Rundensteiner. D-cape: distributed and self-tuned continuous query processing. In *CIKM Poster*, pages 217–218, 2005.

[29] T. Sutherland and E. Rundensteiner. D-cape: a self-tuning continuous query plan distribution architecture. In *Techincal Report, WPI-CS-TR-04-18*, April, 2004.

[30] The STREAM Group. Stream: the stanford stream data manager. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 26(1):19–26, March, 2003.

[31] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[32] T. Urhan and M. Franklin. Xjoin: a reactively-scheduled pipelined join operator. In *IEEE Data Engineering Bulletin*, pages 27–33, 2000.

[33] U.Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, June, 2004.

[34] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[35] S. Zdonik, M. Stonebraker, and M. Cherniack. The aurora and medusa projects. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pages 3–10, 2003.

[36] Y. Zhu, E. Rundensteiner, and G. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, pages 431–442, 2004.

Appendix A

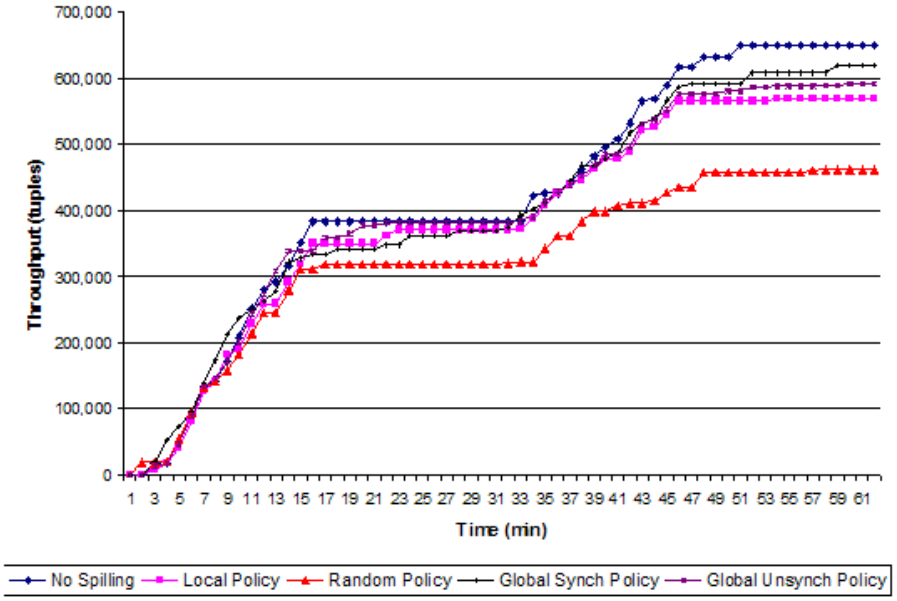

Figure A.1: Throughput, Q^L plan, data set D1. All adaptation policies.

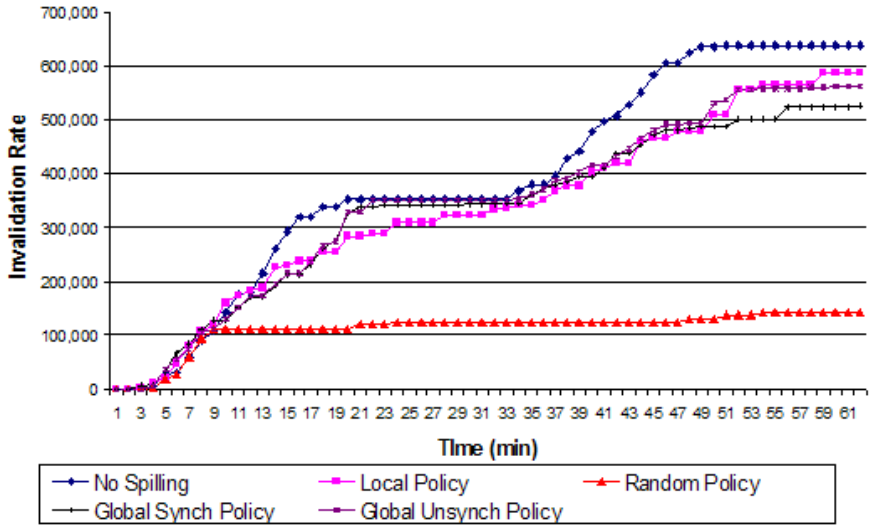

Figure A.2: Invalidation rate, Q^L plan, data set D1. All adaptation policies.

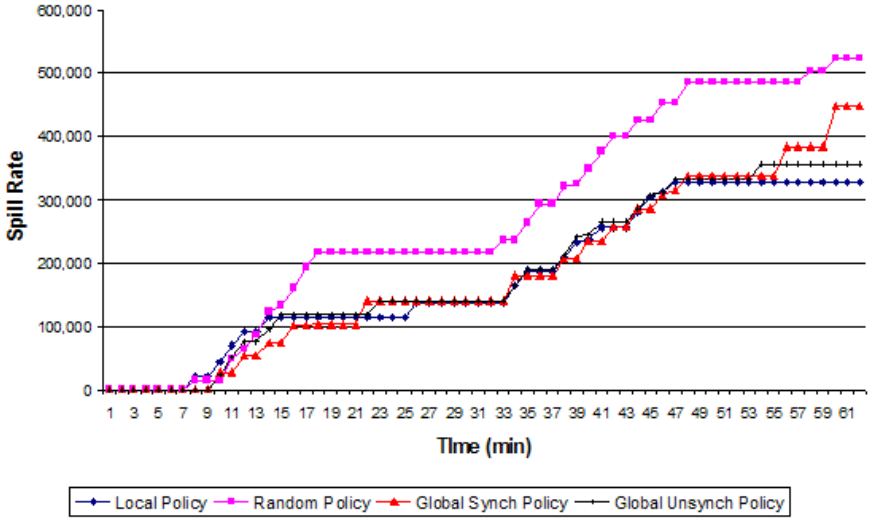

Figure A.3: Accumulated number of spilled tuples, Q^L plan, data set D1. All adaptation policies.

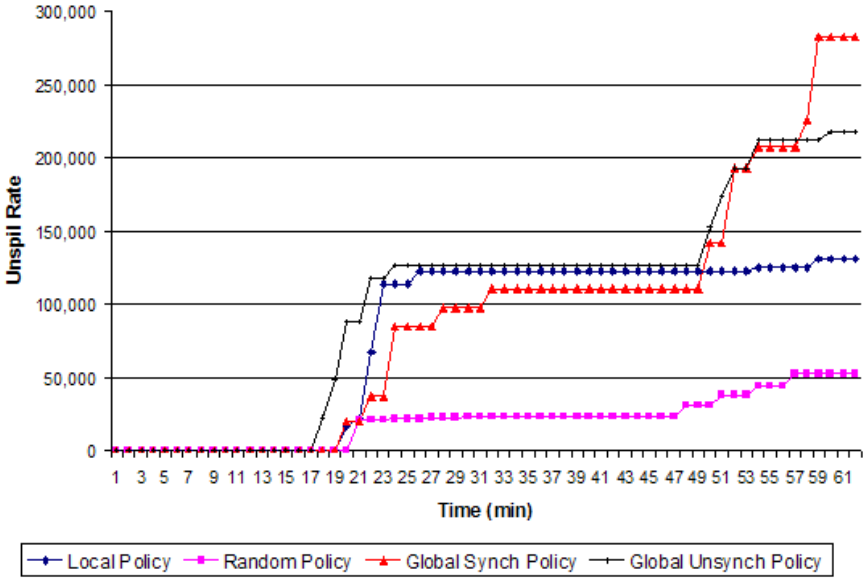

Figure A.4: Accumulated number of unspilled tuples, Q^L plan, data set D1. All adaptation policies.

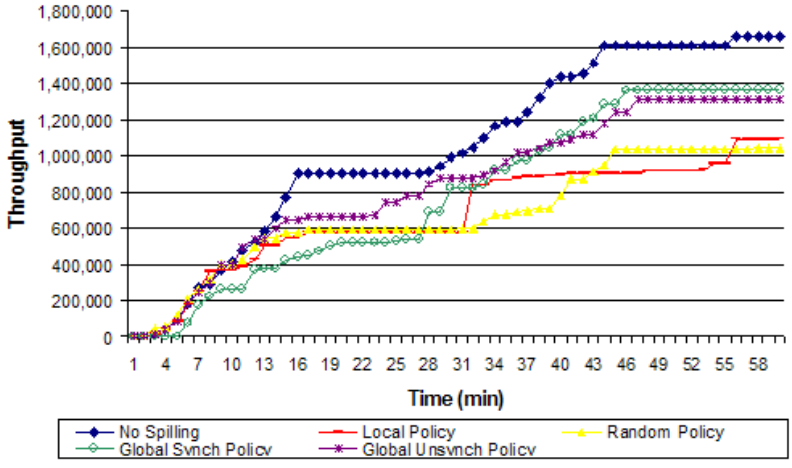

Figure A.5: Throughput, Q^M plan. All adaptation policies.

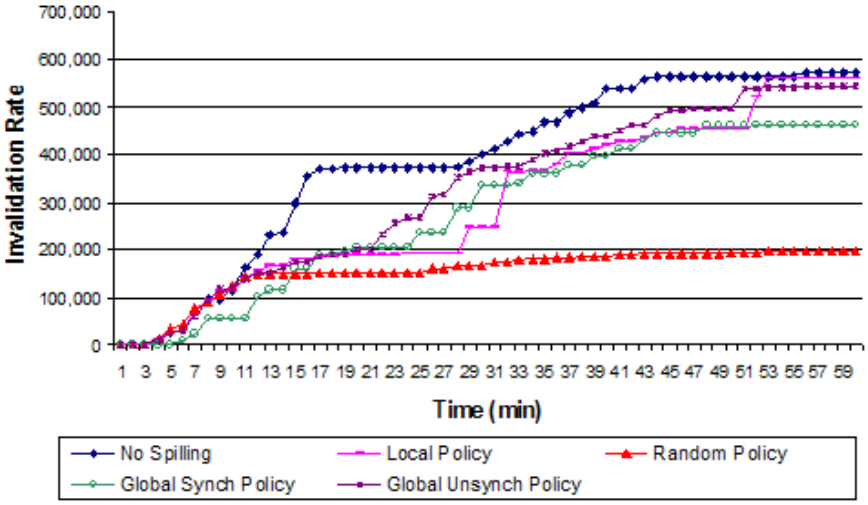

Figure A.6: Invalidation rate, Q^M plan. All adaptation policies.

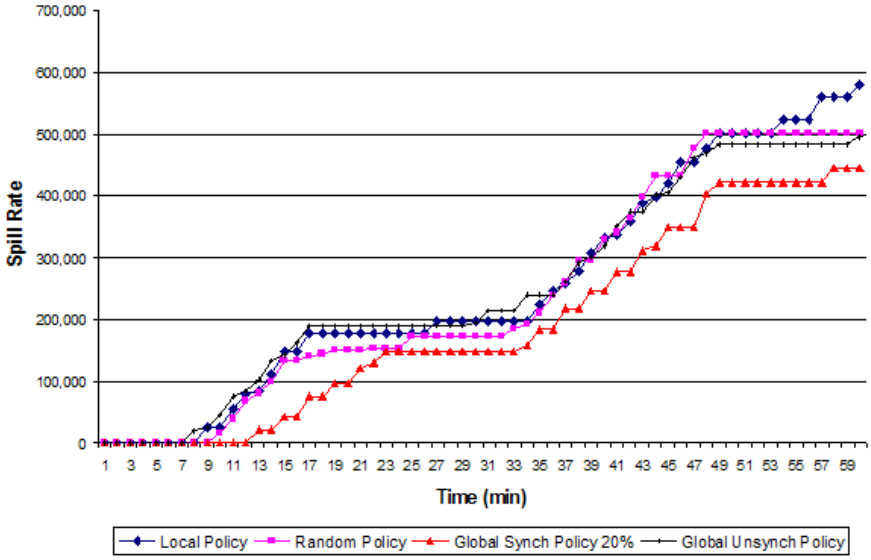

Figure A.7: Accumulated number of spilled tuples, Q^M plan, data set D1. All adaptation policies.

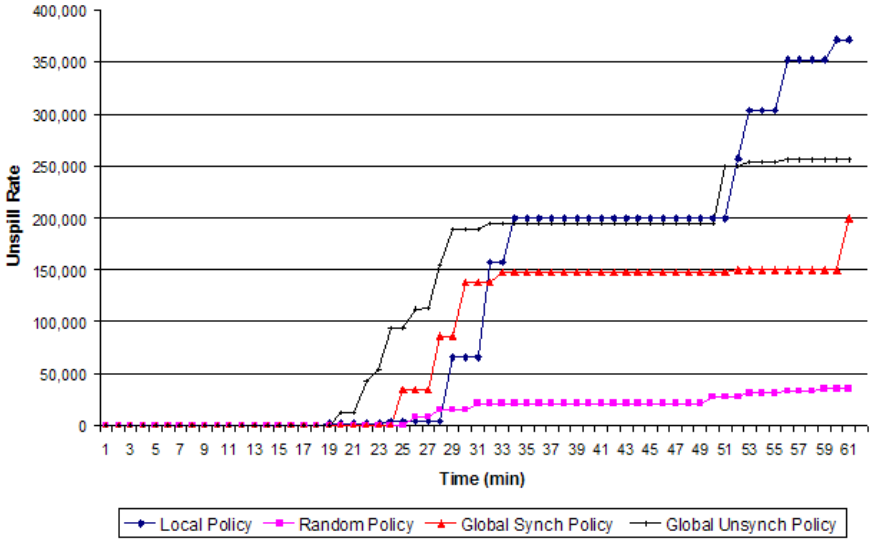

Figure A.8: Accumulated number of unspilled tuples, Q^M plan, data set D1. All adaptation policies.

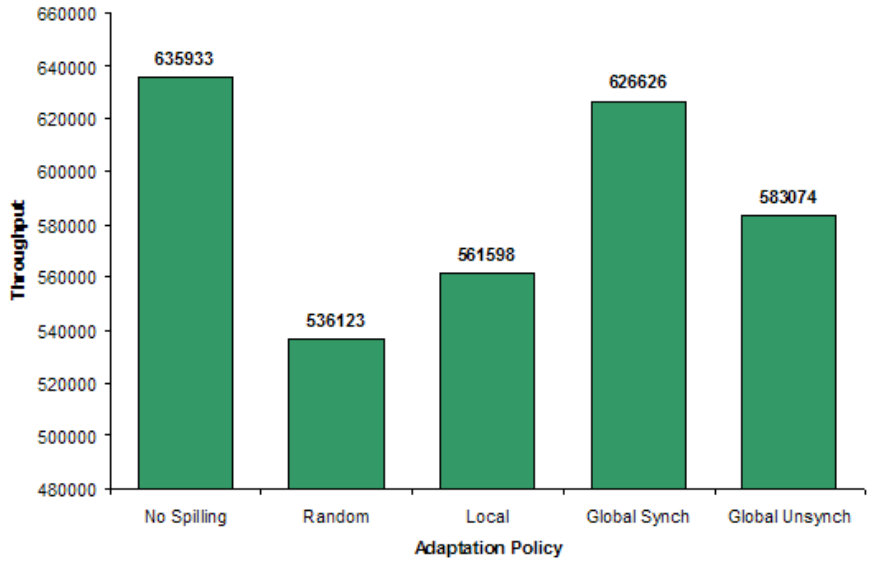

Figure A.9: Throughput, Q^L plan, data set D1, 30/6000 data rate, correlation percentage 5%, dequeue ratio 10. All adaptation policies.

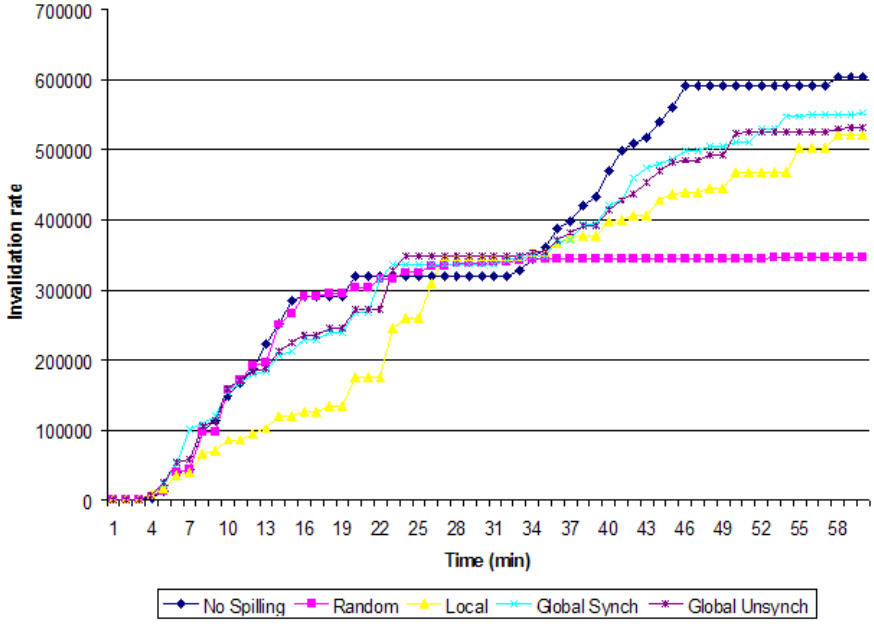

Figure A.10: Invalidation rate, Q^L plan, data set D1, 30/6000 data rate, correlation percentage 5%, dequeue ratio 10. All adaptation policies.