Processing Exact Results for Queries over Data Streams

by

Abhirup Chakraborty

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2010

Abstract

In a growing number of information-processing applications, such as network-traffic monitoring, sensor networks, financial analysis, data mining for e-commerce, etc., data takes the form of *continuous data streams* rather than traditional stored databases/relational tuples. These applications have some common features like the need for real time analysis, huge volumes of data, and unpredictable and bursty arrivals of stream elements. In all of these applications, it is infeasible to process queries over data streams by loading the data into a traditional database management system (DBMS) or into main memory. Such an approach does not scale with high stream rates. As a consequence, systems that can manage streaming data have gained tremendous importance. The need to process a large number of continuous queries over bursty, high volume online data streams, potentially in real time, makes it imperative to design algorithms that should use limited resources.

This dissertation focuses on processing exact results for join queries over high speed data streams using limited resources, and proposes several novel techniques for processing join queries incorporating secondary storages and non-dedicated computers. Existing approaches for stream joins either, (a) deal with memory limitations by shedding loads, and therefore can not produce exact or highly accurate results for the stream joins over data streams with time varying arrivals of stream tuples, or (b) suffer from large I/O-overheads due to random disk accesses. The proposed techniques exploit the high bandwidth of a disk subsystem by rendering the data access pattern largely sequential, eliminating small, random disk accesses. This dissertation proposes an I/O-efficient algorithm to process hybrid join queries, that join a fast, time varying or bursty data stream and a persistent disk relation. Such a hybrid join is the crux of a number of common transformations in an active data warehouse. Experimental results demonstrate that the proposed scheme reduces the response time in output results by exploiting spatio-temporal locality within the input stream, and minimizes disk overhead through disk-I/O amortization.

The dissertation also proposes an algorithm to parallelize a stream join operator over a shared-nothing system. The proposed algorithm distributes the processing loads across a number of independent, non-dedicated nodes, based on a fixed or predefined communication pattern; dynamically maintains the degree of declustering in order to minimize communication and processing overheads; and presents mechanisms for reducing storage and communication overheads while scaling over a large number of nodes. We present experimental results showing the efficacy of the proposed algorithms.

Acknowledgements

I want to thank my supervisor Prof. Ajit Singh for giving me enough freedom is pursuing research and selecting a dissertation topic. This thesis is the result of his constant guidance, advice, feedback and motivation. I would like to thank professor Sagar Naik for keeping his door open for me whenever I needed any advice and suggestion. I thank my PhD Committee members Sagar Naik, Kostas kontogiannis, Kumaraswamy Ponnambalam, Grant Weddell and the external member Luiz Fernando Capretz for their valuable comments and suggestions that significantly improved the technical content of the thesis.

During my stay in waterloo, numerous people have helped me, and I would like to apologize in advance if I miss out mentioning anyone's name. I would like to thank Raihan Al-Ekram, Mohammand Zulkernine, Ziaur Rahman, and Rajesh Palit for their help and support. Raihan Al-Ekram, who was no less than my elder brother, was always there to come up with answers to my questions from arbitray domains. I would like to thank my friends Suman Nath, Al-Hasan, Ali Tareque, Subhashish Ghosh, Ataur Rahim Katebi for thir help, inspiration and support.

I am grateful to my mother and my grandfather for preaching me, during my early childhood, the importance of knowledge and grafting within me the desire to pursue it.

To my grandparents,
Suniti Chakraborty and Gnyanadaranjan Bhattacharjee

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Data Stream Management Systems

The wide deployment of sensor devices, such as wireless motes and RFID (Radio Frequency Identification) readers, and other sources of data in the Internet and the Web have heralded the emergence of a new class of data intensive applications that treats data as continuous streams, rather than a persistent set. The magnitude and nature of the data within such an application pose numerous challenges that can not be addressed by traditional databases. Below, we present some sample applications from diverse domains.

1. Network traffic management involves monitoring network packet header information and network performance measurements across a set of routers and switches. Analysis of such network traffic data presents a strong case for streaming data analysis.

2. Sensor networks, deploying battery powered sensor motes [78, 139, 116, 73] with communication capabilities, involves a technology for collection and real time analysis of information over a large physical world [20, 49]. Various applications for this technology have been proposed: monitoring traffic in highways [46, 45, 114], large scale inventory and supply chain management using RFID tags [143, 62], real time monitoring of environments and habitats [18, 54, 100], structural health monitoring [146, 82, 105], location tracking and surveillance [124, 25, 1] etc. Large scale deployment of these sensor networks will generate immense volume of data in the form of continuous data streams.

3. The volume of data generated from telephone call records, sale transactions, web-server logs etc. is quite large. Online analysis of the transaction logs can provide insight into the

business system; for example, such an analysis can identify interesting patterns in purchases (by customer), can detect and prevent fraudulent activities. In practice, such a real time analysis of the web-logs is critical for the retailers (e.g., Amazon [1], eBay [2]), search engines and web portals to maximize their revenues by giving personalized product recommendations to customers, customizing advertisements, and improving the quality of search results.

4. Within the Web, numerous online data feeds, such as news, sports, financial tickers [93], sensor data [108, 16] etc., produce a large volume of data. A third party service (e.g., Traderbot [3], Sensor-Web [108, 16, 107], HiFi [37] etc.) pulls the (live) data from various sources and processes queries on the collected online data.

In all the applications mentioned above, new data management requirements arise due to continuous, unbounded, rapid and bursty nature of the streaming data. Traditional DBMSs are ill-equipped to support rapid and continuous loading of individual data items, and they do not support the long running, *continuous queries* [132, 12] that are typical of these applications [8]. Therefore, considering the mismatch between the traditional DBMS and the requirements of stream-based applications, a new class of systems—Data Stream Management Systems (DSMSs) [25, 29, 39, 47, 104]—have emerged. In short, these systems aim at processing continuous queries over time varying data streams in real time.

Figure 1.1 shows the abstract architecture of a DSMS. Streaming inputs are stored in the input buffer before being fetched and processed by the query processor. The query processor receives user queries and executes the queries against the online data in real time. The storage manager provides services to manipulate (allocate, deallocate, read, write) data in the disk, and manages the main memory allocated to the DSMS. The disk stores various types of data: (a) tuples from input buffer upon buffer overflow (b) queries and data (state of the queries) upon memory overflow, and (c) other persistent tables or relations. The query processor keeps the necessary cost statistics (arrivals rates, productivity of a query operator, execution time of various operators etc.) to adaptively re-optimize the execution plan, and communicates with both the buffer and the storage manager. The query results are sent to the users as a separate data stream.

[1] www.amazon.com
[2] www.ebay.com
[3] www.traderbots.com

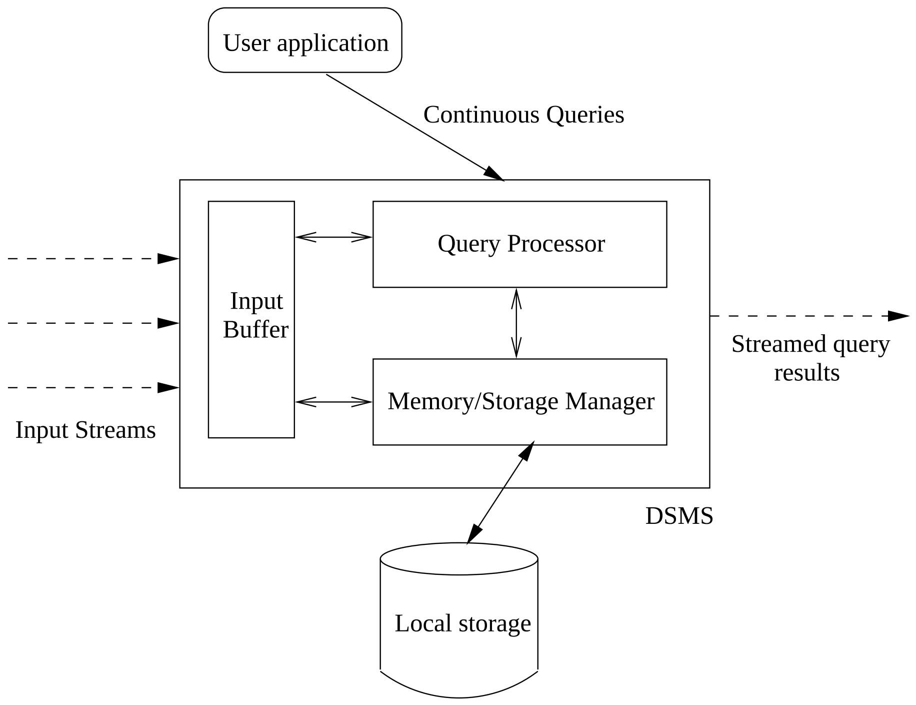


Figure 1.1: Reference Architecture of a Data Stream Management System (DSMS)

1.2 Data Stream Model

In the data stream model, the data arrives in one or more continuous streams of data items, usually referred to as *stream elements or stream tuples*. The input data is not available beforehand to produce statistics on the data. The order of stream arrivals is fixed; and, such an ordering is either implicit (ordered according to the time of arrival at the DSMS) or explicit (ordered according to the generation time—indicated by a *timestamp* attached with each stream element—at the source). A DSMS does not have any control over the arrival order, rate or data distribution of the input streams. As a result, a data stream model has numerous properties and processing requirements differing from the traditional relational model:

- A data stream is a continuous, unbounded sequence of stream elements. Stream elements are usually relational tuples (i.e., *stream tuples*) with predefined structures; but, these could be structureless or semi-structured (i.e., XML documents), video images, or more complex objects.

- A data stream is usually unbounded in size. Thus, storing the entire data stream in memory of disk storage is infeasible. Moreover, stream elements that have arrived recently in the system are likely to be more important to the users. These phenomenons give rise to a sliding-window model.

- Queries over data streams are usually continuous and long running in nature: they are issued once and remain active for a long interval of time. Thus, upon the arrival of a stream element, new results for the queries should be evaluated in real time, i.e., provide continuous results.

- The system conditions (e.g., arrival rate, data distribution, query loads, available memory) may vary significantly during the lifetime of a continuous query. Thus, a *plan-first execute-next* approach [121] to process queries is no longer effective in a data stream model, and an adaptive approach to query processing is necessary [13, 9, 77, 5].

Requirement 2, as stated above, implies that some continuous queries over unbounded streams are not computable using finite memory (e.g., a join operator). Also, blocking operators (e.g., sort, sum, count, max etc.) have to scan the entire input before generating an output tuple. One popular technique to solve these problems in data stream processing is to evaluate the queries over a sliding window of the stream. For example, data from the last week could be used in evaluating a query, thus discarding data older than a week. The sliding window expires old stream elements as new ones arrive. Sliding window based approximation, though mitigates the problems related to memory requirements and blocking of operators, gives rise to additional issues. To realize the challenges of processing queries over sliding windows, consider a simple query to compute the maximum value over a data stream. Such a query, though trivial in an arbitrarily large stream, requires $\Omega(N)$ space while evaluating over a window of size N; in case of a stream with non-increasing values (so, the oldest item is the maximum), the maximum value changes every time the window slides forward.

1.3 Problems

Processing continuous queries over data streams with a high arrival rate imposes high resource requirements on a Data Stream Management System. State-based queries (e.g., stream joins) with large state sizes requires enormous memory footprints for their execution. Moreover, in a DSMS, that processes numerous concurrent queries, the available resources for an individual query is limited. In addition, the arrival rates of the input streams are bursty and unpredictable; thus, estimation of the peak arrival rate for an input stream in infeasible. Even if such a peak load is known beforehand, provisioning resources to handle the peak load is not resource efficient. This is due to the fact that the peak load is usually an oder of magnitude higher than the average load, and hence provisioning resources for the peak load would leave most of the resources idle during normal operations. Processing stream queries in a resource limited system brings forth numerous technical challenges. In the subsequent part of the section, we present the issues associated with the resource limitations of a DSMS. Developing techniques to cope with these issues is the central focus of this thesis.

1.3.1 Disk-based query processing

Resource limitations in a DSMS along with its high stream arrivals that exhibit temporal fluctuations motivated numerous research works to devise resource allocation strategies that shed loads during periods of high loads. Such load shedding algorithms provide approximate results and save resources (i.e., CPU and memory) by dropping tuples, which impart adverse effects on the accuracy of state-based queries (e.g., stream join). As in traditional relational database systems, a join operator is a very important and resource intensive operator in a DSMS, that evaluates stream joins over a sliding window. Given a limited memory for a sliding window join, no online strategy based on load shedding can be k-competitive for any k that is independent of the sequence of tuples within the streams [125]. In such a scenario, generating exact results for the stream joins should incorporate a disk storage to spill overflowing states; also, for a system with QoS-based query output, secondary storage is necessary even to guarantee a QoS above a certain limit.

In such a system, the query processing systems should operate in cooperation with a disk archive to address the problem of run-time memory shortage while generating complete result sets of a stream join. Instead of dropping the stream tuples during high stream load, the query system must push the operator states or stream tuples to the disk. The system must reduce small and random disk I/Os (*reads* or *writes*) while integrating the disk resident tuples. The need for expiring tuples as the window slides may impose a considerable random I/O load on the system. Maximizing output generation rate, an important issue in an online processing environment (e.g., [72, 119, 69]), remains to be a key issue in processing stream joins.

The challenge at the query system, therefore, is to devise a join processing algorithm that (a) should be I/O efficient, (b) should render the disk accesses largely sequential eliminating small, random disk I/Os, and (c) should maximize output generation rate reducing the average delay in generating an output tuple.

1.3.2 Processing hybrid queries

Processing hybrid queries, that involves both the streaming and persistent (or historical) data, are widely used in numerous applications like network traffic analysis, financial data analysis, supply chain management, active data warehouses etc. [31, 115]. In an active warehousing, joining an update stream with a persistent relation is the key operator to a various transformations (e.g., coded value translation, surrogate key generation, duplicate detection etc.) used in the Extraction-transformation-Loading (ETL) process. In such an environment, the join operator should not be resource intensive, since the transformation operator executes concurrently with other transformations in a pipeline and the transformation should not block the normal system activity. Moreover, processing a hybrid join operator is challenging due to the asymmetry or mismatch in the input access costs: the tuples from input streams arrive at a high rate, whereas tuples from the disk relation are costly to retrieve due to expensive disk I/Os.

1.3.3 Parallelizing query operators

As observed above, the stream applications place several scalability challenges on the system. With the increase in the bandwidth of the networks linking together processing nodes with a broad range of processing capabilities and memory sizes, distributed stream processing applications are becoming increasingly common. Scalable processing of data streams over a distributed system become more and more challenging with the increase in the volume of online data generated by the applications. In such an environment, a processing node can be shared by multiple applications; therefore, over-provisioning for the peak load of any application is, if feasible, not cost-effective. Also, non-query background loads and the available memory vary on each of the nodes. In such a system, diffusing loads of a window join across a number of independent nodes is a challenging problem. Such an intra-operator parallelism should minimize both the communication overhead incurred within the network and the processing overhead within the processing nodes; the system should dynamically adjust the join loads across the processing nodes, and should dynamically vary the degree of declustering in order to optimize the (communication and local processing) overheads.

The work presented in this thesis addresses the main technical challenges associated with scalable processing of stream join queries in a resource-limited DSMS.

1.4 Contributions

To summarize, the following are the main contributions of this dissertation:

- First, we identify the importance of processing exact results for sliding window join queries over data streams using limited memory. We propose a brute force algorithm, called Exact Window Join (EWJ), that processes stream join queries incorporating secondary storages; the proposed algorithm eliminates random disk I/Os and exploits the high disk bandwidth by rendering the disk access patterns largely sequential.

- Second, we propose a hash-partitioned join algorithm that dynamically adapts the memory allocation at the levels of *stream windows* and partitions within a window. The proposed algorithm minimizes random partition switches, while joining with the disk resident data, by properly arranging blocks during a disk-dump. We extend an existing algorithm, that joins unbounded streams, to devise a baseline algorithm *Rate-based Progressive Window Join* (RPWJ) to process sliding window joins. We demonstrate the efficiency of the proposed algorithm based on the comparison with the baseline algorithm.

- Third, we propose a partition-based algorithm to process hybrid join queries spanning both an update stream and a persistent relation. The proposed algorithm exploits spatio-temporal locality within the update stream and improves output delay by exploiting the hot-spots in the range or domain of the joining attributes. We develop techniques to amortize a disk access over a large number of stream tuples. We propose the *multi-granularity processing* technique to reduce processing overheads by dynamically switching the processing granularities.

- Fourth, we propose techniques to parallelize a stream join operator over a shared nothing cluster. We develop techniques to diffuse and adapt loads across the processing nodes, to fine tune the partitions to reduce CPU overheads. To solve the scalability issues in a large scale parallelism, we propose the *adaptive declustering* technique to adapt the degree of parallelism based on the join loads, and the *subgroup communication* technique to control the communication overheads.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 introduces some of the preliminaries and discusses the work related to data stream management systems in general, and data stream join processing techniques in particular. The main focus of this chapter is a discussion of various techniques related to processing stream joins, hybrid joins and distributed stream query processing. Chapter 3 focuses on a brute force method for processing window stream joins. This chapter presents an algorithm for processing exact results of a stream join incorporating a secondary storage. Chapter 4 outlines the necessary changes to an existing algorithm to process joins over bounded streams RPJ, a variant of XJoin algorithm [133], and presents an algorithm called RPWJ (rate-based progressive window joins) for processing windowed stream joins. The RPWJ algorithm serves as a baseline algorithm in the subsequent chapter. Chapter 5 presents a join algorithm, called Adaptive Hash-partitioned Exact Window Join (AH-EWJ), for processing windowed stream join with limited memory. This chapter presents a number of techniques for minimizing disk and CPU overhead, adapting sizes of both the stream windows and the partitions, maximizing the output generation rate of the join query. Chapter 6 considers a hybrid join between a live stream and a persistent table, and provides a partition-based algorithm processing the hybrid join. Such a hybrid join is widely used in various transformations (e.g., surrogate key generation) used in active data warehousing. Chapter 7 provides a framework for parallelizing a stream join operator within a shared nothing cluster. This chapter provides techniques to diffuse loads across the non-dedicated processing nodes, to dynamically determine the degree of parallelism, to reduce CPU overhead by fine tuning the partitions within the processing nodes, and to remove a scalability bottleneck by providing a fixed or predefined communication pattern among the processing nodes. Chapter 8 concludes this dissertation, and presents a discussion of possible directions for further research.

Chapter 2

Background

In this chapter, we present an overview of the works on data streams related to this thesis. This chapter is by no means a thorough survey of all the work on data stream processing, and the reader is referred to the survey articles [8, 106, 60] for further information and pointers. In what follows, we present a brief overview of data stream models, provide motivating examples of join processing in a DSMS, present prominent system research aiming at building systems for processing data streams, present related works on processing join queries over data streams.

2.1 Preliminaries

2.1.1 Data Models

Several stream models have been proposed for data stream processing. In most of the models, items within an input stream $a_1, a_2 \ldots, a_i, \ldots$ arrive sequentially (or, in a bursts) and describe an underlying *signal*. The signal can be represented as a function $A : [0, \ldots (N-1)] \rightarrow \mathbb{R}$; the domain of the function A is assumed to be discrete and ordered, the function maps it to non-negative integers. A value i within the domain of A is usually thought of as a discrete time index, and therefore a_i is denoted as the element arriving in the stream at time i. The Stream item may describe the underlying signal in one many ways, yielding different data models. Each stream item may arrive out of order and/or in pre-processed form. For example, in network traffic management, rather than generating one stream element for each IP packet, one element (or several partially pre-aggregated ones) may be produced to summarize the length of connection between two IP addresses and the total volume of data transmission. This gives rise to the following models [57]

- *Unordered cash Register*: Stream items arrive in no particular order on the domain values, and without any pre-processing. This is the most general model.
- *Ordered Cash Register*: Stream items are not pre-processed but arrive in the order of the domain values (e.g., timestamp).
- *Unordered Aggregate*: Stream items are pre-processed, and only one item per domain value arrives in no particular order of the domain values, e.g., one stream item per TCP connection.
- *Ordered Aggregate*: Stream items are pre-processed and one item per domain value arrives in a particular order, e.g., one stream item per TCP connection arriving in increasing order of the connection end-times.

In most applications, without loss of generality, the entries of A can be assumed to be integers. We make this assumption in the remainders of this dissertation for the sake of simplicity. However, all the above models can be generalized to multidimensional data, where the entries of A represent points in multidimensional space.

Additionally, in many practical applications, the stream arrivals and the distribution of tuple values are bursty or skewed in nature [84, 85, 92, 150]. Moreover, the data streams exhibit spatio-temporal locality of the stream tuples: values of one or more attributes of the stream tuples arriving at time near t might be close to each other within the domain of the attributes [91, 95, 137].

2.1.2 Stream Windows

As discussed in Section 1.2, a stream window serves as the mechanism to limit the scope of a query operator to recently arrived tuples. Such a window-based query evaluation reduces the memory pressure on the system, as the stream elements falling outside the sliding window can be expired, freeing up the memory. Emphasizing on recent data is relevant and important to a majority of real-world applications. For this reason, the notion of windows is incorporated in various declarative query languages, such as CQL [4], GSQL [39] etc. Windows can be classified into various types according to different criteria. Here we outline the most popular ones:

1. *Movement of window endpoints*: A window with fixed beginning and end points is known as a *fixed window*; a window where both the beginning and end points move,

replacing old tuples with the arrival of new ones, is called a *sliding window*; a window with a fixed beginning point and a forward moving end point is known as a *landmark window*.

2. *Specification of a window*: A *time-based* window is defined in terms of a time interval, whereas a *count-based* window is defined in terms of the number of tuples within the window. In a *predicate window*, an arbitrary predicate specifies the content of the window [55]; for example, all temperature sensors reporting temperatures above 20 during last T time units.

2.1.3 Stream Processing

Data management systems can be classified according to their ability to support different types of queries over data streams. Queries over data streams could be classified into different categories depending on different aspects. Considering the temporal aspects, the queries can be of two types: *one time or snapshot queries* and *continuous queries*. Snapshot queries, that are similar in nature to the traditional DBMS queries, are evaluated once over a snapshot of the data set at a particular instance.

Continuous queries—typical of data stream applications—are evaluated continuously with the arrival of new stream elements. These queries usually run for a long interval of time and return results to the users during the life time of the queries. The continuous queries generate incremental results upon the arrival of new stream tuples, or with the passage of time. In a DSMS executing continuous queries, the arriving stream tuples are directly sent to the query engine. Results from continuous queries may either be stored and updated with the arrival of new data, or be produced as data streams themselves [8]. For example, aggregation queries may involve updating the result frequently, thus suggesting an approach with result store. Whereas, the join queries may produce monotonic, unbounded elements, suggesting the stream approach.

Continuous queries can be classified into two categories based on the nature of their inputs: those accessing only live streams as the inputs, and those with both live streams and persistent tables as the inputs. We refer to the continuous query with all the inputs being live streams as *continuous stream queries* (or *stream queries*, for short). On the other hand, We use the term *continuous hybrid queries* (or *hybrid queries*, for short) to refer to the class of continuous queries that requires to access both live streams and persistent tables.

Depending on the composition or declaration, continuous queries can be classified as *predefined* and ad hoc queries A predefined query is supplied to the DSMS management system before system initiation. Ad hoc queries are issued online as the DSMS is active and data streams have begun to stream.

2.2 Motivating Examples

In this section, we present the examples of continuous queries over data streams as found in many application domains such as finance, web applications, network monitoring, sensor monitoring. These examples motivate the data stream join operators that are widely used in a DSMS. We present examples to provide motivation for both *stream queries* and *hybrid queries*.

2.2.1 Stream Queries

In a *network Traffic Management* domain, which involves analyzing network packet traces collected from a number of links in a large network such as the backbone of an ISP [50, 39], a network analyst poses queries to detect service-level agreement violations, to monitor network health, or for other reasons [8]. In this setting, let us consider the following query:

> *Monitor total traffic from a customer network that went through a backbone link of an ISP within the last one hour.*

Let C denote a customer link that connect the customer network to the ISP's network. Let B denote an important backbone link connecting two routers of the ISP's network. Data collection devices on these links collect packet headers, process the headers (e.g., to compute packet identifiers [44]) to generate stream tuples, and send the streams to the DSMS running the continuous queries. Thus, we have two input streams which, for convenience, are also denoted as B and C. Each tuple in these streams contain a packet identifier pid and the packet's size in bytes. The continuous query mentioned above can be posed in CQL [4] as:

```
Select   sum(C.size)
From     C[Range 1 hr], B[range 1 hr]
Where    C.pid=B.pid
```

This continuous query first joins streams B and C on the join attribute pid using a sliding window of length one hour. The output of the join is then aggregated to continuously compute the total traffic common to both the links. A similar query can be used in sensor network, e.g., to track moving objects [70].

In an *online auction system* [131], people continuously submit new items for auction, and bids arrive continuously for the items. Consider an auction system with the following streams:

S_1: OpenAuction(item-id, seller-id, start-price, timestamp)
S_2: Bid(item-id, bid-price, bidder-id, timestamp)

When a seller starts an auction, a tuple arrives in stream S_1. On the other hand, when a bid is placed on an auction, a tuple arrives on stream S_2. Consider the following query on the system: *for each seller, find the total bids received on all his auctions in the last 4 days.* This query requires a sliding window join between streams S_1 and S_2, followed by an aggregation operator. Here, the window-size of stream S_2 is four days, and that of stream S_1 is determined by the maximum lifetime of an auction.

As an another example, consider an order management system with the following schemas of the streams:

S_1: Orders(order-id, customer-id, cost, timestamp)
S_2: Fulfilments(order-id, clerk-id, timestamp)

In such a system, consider the following query: *find total cost of orders fulfilled by each individual clerk over the last 12 hours, and update the result periodically (every one hour).* In CQL, the query can be given as follows:

```
Select      F.clerk-id, sum(O.cost)
From        Orders O[Range ∞], Fulfilments F[range 12 hour,
                                           slide 1 hour]
Where       O.order-id=F.order-id
Group By    F.clerk-id
```

There are a plethora of real-life applications involving stream joins: finding similar news items from different news sources; finding correlation between phone calls and stock trading [52]; processing data streams to detect complex events in retail management, healthcare [143] etc. The Stream Query Repository [131] contains further examples of queries involving stream joins in various application domains, e.g., Online Auctions, Network Traffic Monitoring, Military Logistics, etc.

2.2.2 Hybrid Queries

In an active data warehouse, updates are propagated to the warehouse in an online fashion (i.e., as quickly as possible) through an ETL (Extraction-Transformation-Loading) process. In an ETL process, the transformation of update tuples is the main operation, which requires significant processing and disk I/O loads. In such a scenario, joining an update stream with a persistent table is central to numerous transformations such as surrogate key generation, duplicate detection, coded-value translation, etc. In Chapter 6, we provide an example of such a transformation operation used in surrogate key generation.

Hybrid queries are relevant to various other applications. Here, we provide an example of queries frequently observed in financial monitoring or stock trading. Consider a scenario where a trader poses continuous queries over streams of trading data from stock markets. An example of one such query (as taken from [31]) is shown below, which *monitors the total amount of money from transactions of stocks with beta coefficient exceeding 1.05* [1].

```
Select     T.Symbol, sum(T.volume*T.price)
From       Trades T[Range 30 minutes, Slide 5 minutes],
           Information I
Where      (T.Symbol=I.Symbol) and
           (I.beta >1.05)
Group By   T.Symbol
```

The query maintains the result over the stock items transacted within a time interval (the last 30 minutes), and updates the result periodically (every 5 minutes). The stream Trades is joined with a persistent table (Information) using the join predicate (T.symbol=I.symbol). In the query, the predicate (I.Information>1.05), that involves only the attribute of a single table, is known as an individual predicate.

The following example provides another query that accesses both the real time (streaming) and historical (stored) data in a uniform manner [26].

```
Select     T.Symbol, count(*) as Swings
From       Trades T, TradeStore TS
Where      (T.Symbol=I.Symbol) and T.Date-TS.Date ≤ 1 Year
           and (TS.Fluctuation ≥ 0.05)
Group By   TS.Symbol
Having     Swings ≥ 6
```

[1]The beta coefficient of a stock or portfolio is a number describing the relation of its returns with that of the market as a whole

This query joins the stream Trades with an archive that stores trade-tuples appeared in the last one year. The query returns the stream tuples for the trade items for which the $\mathrm{Fluctuation}$ index crosses a threshold (0.05) a number of times (6) within the last one year.

In all the queries stated above, the core operator is a *stream join or hybrid join* that is, as for traditional database systems, a very important operator in a DSMS. Stream or hybrid joins are necessary whenever information from several sources (live streams or stored table) should be combined to compute correlations or to match events.

2.3 Data Stream Management Systems

Having outlined the join queries to be supported by a stream processing system, we turn to Data Stream Management Systems that have been proposed to enable such queries.

From a systems point of view, a key challenge for a Data Stream Management System is to replace the pull-based iterator model of processing queries, as observed in a traditional DBMS, with a data-driven, non-blocking, push-based model. Such an issue has resulted in the redesign of relational operators, so that the stream operators becomes non-blocking and can return incremental results; also, new stream-specific operators such as *sliding windows* need to be designed within such a Data Stream Management System. Designing a data stream management system (DSMS) requires substantial modification of almost every aspect of data management. Several issues in a DSMS such as timestamps, ordering and the sliding windows complicate the data model and query language.

Despite the above common requirements, the stream projects at different research organizations have taken different approaches. Consequently, the challenges in data stream processing have spawned a vast number of full-fledged data stream systems such as as Aurora [25], Hancock [38], Gigascope [39], Tangram [111, 112], NiagaraCQ [32], OpenCQ [97], and others. In the subsequent part of the section, we present the recent stream projects that have been conducted at different academic institutions, starting with the early, first generation systems for processing continuous queries over data streams.

2.3.1 First Generation Systems

In the field of processing continuous queries over data streams, two projects, Tapestry [132] and Tribeca [127], have made seminal contributions. These projects can be thought of as first generation continuous query systems that served as precursors to the data stream management projects, that have gained prominence in the research community.

Tapestry system first proposes the continuous queries for filtering streams of electronic documents, such as mail messages or news articles. The tapestry system maintains information about a document, such as its authors, date, keywords, title etc., in an append-only database: new documents are added to the database as they arrive and are never removed. In Tapestry, queries are written in a SQL-like language called TQL. The system proposes a clean, time-independent semantics for continuous queries, as simple periodic execution of a continuous query at a certain interval might yield non-deterministic results. The continuous semantics, the desired semantics for continuous queries, are defined as follows: "the results of a continuous query is the set of data that would be returned if the query were executed at every instant in time." In other words, computing results of a TQL query against an append-only database can be considered as a two-step process. In the first step, a one-time SQL query is executed over a snapshot of the database at that time instant. In the second step, the results of all one-time queries are merged and returned to the user. For performance reasons, a continuous query is rewritten into an *incremental* query that should only examine newly arrived data.

Tribeca is an early continuous query system, that provides a query language [128], and supports applications for monitoring network logs. The Tribeca system identifies a set of stream processing operators, and converts a query into a network of unary dataflow operators; each of the operators takes precisely one stream as an input, and produces one or more output streams.

Both the Tapestry and Tribeca systems have shortcomings. The Tapestry language (TQL) does not support either aggregates or sliding window join operations. Though Tribeca query language supports sliding window operations, it does not support join operations since Tribeca only supports unary stream operators. Also, tribeca operators cannot handle historical or persistent data.

Despite these shortcomings, both Tapestry and Tribeca systems are significant projects, and are considered the intellectual forebears of many significant research projects. In the remaining part of this section, we explore some of these prominent systems.

2.3.2 Aurora

The Aurora system [25] supports monitoring applications that require the rethinking of fundamental architecture of a DBMS. These applications execute a large number of continuous queries over continuous data streams. The Aurora system aims at developing a data stream management system that is scalable with both the number of queries and the stream rates. The core of the Aurora system consists of a large network of triggers (or queries) where each query is a data-flow graph with each node being an operator (or box); stream queries are specified through a *boxes and arrows* paradigm, where different stream operators (boxes) are connected to each other through directed queues (arrows). In practice, the data-flow or the query plan is specified manually by the user.

For each monitoring application, the users create and add one or more queries in the aurora query-network. Aurora performs both compile-time and run time optimization of the query network. This system allows the users to specify quality-of-service (QoS) requirements for monitoring queries, and uses these service requirements to make decision, during run time, on operator scheduling [24] and load shedding. The load-shedding solution [25] involves a careful examination of the query network to select a number of strategic points in the network where the system drops a fraction of tuples during high loads, as is the case when there are sudden bursts of input data.

Aurora supports a rudimentary form of archiving historical data: data flowing throw any of the arrows (or, connection points) in the query network can be archived by specifying the history requirement at the respective connection points. The archives can only be used for historical queries. Aurora does not support hybrid queries that combine live and archived data.

2.3.3 STREAM

The STREAM project [8] at Stanford aims to build a general purpose data stream management system that supports a declarative query language called CQL [4]. STREAM is a relation-based data stream management system with an emphasis on memory management and approximate query processing. In particular, one of the goals of the project is to understand the memory requirement of various queries, and to devise techniques to run queries efficiently within a bounded amount of memory, minimizing the memory profile of a streaming data flow [7]. The CQL language, one of the major contributions of the STREAM project, formalizes the semantics of a stream query processor, and provides mechanisms for defining various windowed queries.

2.3.4 TelegraphCQ

TelegraphCQ [29] system, developed at UC Berkeley, focuses on handling a large volume of continuous queries over high volume, highly-variable data streams. It aims to build a highly adaptive stream-processing system to process data from sensor networks and Internet sources. Streaming nature of the input data in the target applications demands a push-based query processor, invalidating a pull-based one (in traditional database systems) that operate by pulling data from the disk (e.g., using an iterator model). TelegraphCQ uses a continuously adaptive query processing technique, called CACQ [99], that shares work across multiple continuous queries. The push-based infrastructure is based on Fjord modules [98]. Also, the CACQ engine, that does not have any fixed query plan, optimizes the query execution cost using an Eddy [5] that controls the order and route of a stream element across the query operators. With changes in any of the stream rates, data distributions, and the pool of continuous queries (due to insertion or deletion of queries), the Eddy operator continuously adapt the query execution by adjusting the dataflow across the query operators.

2.4 Join over Bounded Streams

Existing join algorithms on streaming data can be classified into two categories: the first one considers bounded or finite size relations, whereas the second category considers the streams of infinite size. The *finite stream joins* focus on generating progressive results. In case of the second category, the tuples are joined based on sliding windows that span a time interval in a recent past of the data streams. This section presents research works on processing joins over bounded streams, where an overview of the work on windowed joins over unbounded streams is presented in the subsequent section.

Unlike traditional databases where all the tuples are available before processing a query, queries on data streams are processed as stream tuples arrive from the remote sources through an underlying network. Thus, traditional join algorithms [63, 101, 123] employing a pull-based model are inapplicable in a push-based scenario (i.e., data streams). The major challenge lies in the blocking of a data stream due to the burstiness in the stream generation or due to the network failures (i.e., congestion, packet loss etc). In addition, generating first few tuples as quick as possible is imperative for any interactive or pipelined execution [72, 90]. In this section, we focus on the push-based scenario, and present significant algorithms developed for processing joins over bounded or finite streams.

There exist numerous join algorithms for joining streaming finite data sets in a non-blocking fashion. Symmetric Hash Join [141], that extends the traditional hash join algorithm, is the first non-blocking algorithm to support pipelining. This algorithm maintains in memory two hash tables with m buckets for sources A and B. Once a tuple t arrives in a source, for example, A, the bucket $h(t)$ of source B is probed, and tuple t is stored in the hash table of source A.

In a symmetric hash join algorithm, the two relations should fit in memory; and this constraint leads to thrashing for larger input relations or streams.

The XJoin algorithm [133] extends the symmetric hash join algorithm to support joins where the available memory is not enough to hold both the input relations entirely. Incorporation of a secondary storage creates two challenges: flushing partitions to the disk and eliminating duplicate results. XJoin employs a simple flushing policy: when memory gets filled, the XJoin algorithm flushes onto the disk the largest hash buckets among the relations A and B. When any of the sources is blocked, the XJoin uses the disk resident buckets in processing joins. The XJoin proceeds in three stages. The first stage, similar to the original symmetric hash join, joins memory resident tuples; this stage is called *mm-stage*. The second stage, called *md-stage*, is activated whenever both the input streams are blocked. This md-stage selects a memory resident partition and joins the partition with the disk resident partition from the opposite stream. The third stage, called the *dd-stage*, is executed after all tuples from both the streams have been received. This clean-up stage ensures that the join results are complete (i.e., all the tuples that should be in the result set are generated eventually). This stage is necessary because the mm- and md-stage may only partially compute the results. On the other hand, duplicate elimination is necessary during md- and dd-stages, as they may perform overlapping tasks producing spurious duplicate tuples. XJoin uses duplicate prevention mechanism based on timestamps.

In reference [135], the authors present multi-way join (MJoin) operators, and claims performance gain while compared with any tree of binary join operators. In case of the multi-way join, arriving tuples in any source may generate results in a single step without waiting: no intermediate results are produced or passed unlike the multistage binary pipeline. Also, there is no need to restructure the plan with the changes in data arrival pattern. MJoin flushes partitions onto the disk using an approach called *coordinated flushing*. This flushing policy spills data from a coordinated set of partitions. That is, if data is flushed from a partition p of a stream, subsequent flushes select data from partition p of all other streams; a new partition is selected only when all the tuples in partition p of all the streams are flushed. The absence of intermediate tuples causes processing overheads in MJoin operators. Adaptive caching [11] considers the problem of adaptively placing and removing caches, that materialize the intermediate sub-results, to optimize the performance of an MJoin. Reference [10] focuses on the problem of ordering pipelined filters, which is common in stream applications and captures a large class of multiway joins. The algorithm determines the order of execution of a number of pipelined filters adaptively to minimize processing cost in an environment, where stream characteristics vary unpredictably over time.

Progressive-Merge Join (PMJ) algorithm [42, 43] is the non-blocking version of the traditional sort-merge join [19]. The sort-merge join first sorts both data sets and then scans, in a merge like fashion, through both the sets. Such a technique starts producing first results only after a considerable time period has been passed. PMJ eliminates the blocking behavior of a sort-based join algorithm. The basic idea of the PMJ is to produce output tuples of the join in parallel to sorting the input sets. PMJ divides available memory into two partitions, one for each source; and the algorithm runs in two phases. The first phase creates sorted subsequences (or *runs*). In this phase, PMJ reads data from the inputs until the available memory is full. Both subsets are sorted, and joined using an in-memory join algorithm. The sorted subsequences (or runs) are then flushed into the disk. In the second phase, when all data is received, PMJ generates longer runs by merging the subsequences that were flushed onto the disk. The runs obtained from merging are again written to the disk. One of the important ideas of PMJ is the generation of output tuples during a merge.

The Hash-Merge-Join (HMJ) [102] algorithm combines advantages of XJoin and PMJ in an attempt to achieve three goals: (1) minimize the time to produce first few join tuples (don't wait till the memory is full), (2) produce join results even if the sources of the join operators are blocked (always keep a portion of data in memory), and (3) minimize the total time to calculate the join (selective flushing). The HMJ algorithm works in two phases: The hashing and the merging phases. Incoming tuples are stored in in-memory hash buckets based on their hash values. The input tuples are joined with the relevant in-memory tuples during the hashing phase. When the memory is full with the incoming tuples, a portion of the memory is flushed onto the disk. The flushing policy aims at balancing the memory across the two input streams by selecting a pair of buckets, one from each of the sources. In merging phase, buckets stored in the disk during hashing phase are retrieved; the disk-resident buckets are then merged, similar to the second phase in PMJ, and are stored in the disk as lager subsequences or runs. The merging phase starts when both the input streams are blocked. If the blocking of the streams are resolved, the hashing phase starts again. HMJ alternates between hashing and merging phase till the whole data in both the streams are processed. Then, the merging phase cleans up the disk-resident runs to generate the remaining output tuples. Duplicate elimination is HMJ is based on associating a block-id with every block: in merging phase, tuples from the similar blocks (i.e., from the same tun) are not joined.

RPJ (rate-based progressive join) [129] algorithm extends the XJoin for progressively joining stream relations. Compared to the MJoin algorithm, RPJ considers input arrival rates while making the flushing decision. Also, RPJ invokes reactive stages (md- or dd-stages) based on the output generation rate of each of the stages. Based on a probabilistic study of the input stream statistics (data distribution, arrival patterns, etc.), it develops a flushing policy that selectively keeps partitions in memory with a view to maximize the output rate.

RPJ switches between the mm-stage and the reactive stage depending on, respectively, the blocking and the resolution of the blocking of the input streams. In the mm-stage, the incoming tuples are mapped to its partition based on a hash function, and joined with the respective partition from the opposite stream. Upon memory overflow, the algorithm flushes a predefined number of tuples, given by a system parameter N_{flush}, onto the disk. The flushing policy selects the victim partitions, based on the arrival probability of tuples within a partition, until the desired N_{flush} number of tuples are flushed onto the disk. The flushing algorithm sorts the partitions in an ascending order of the values of the arrival probabilities of tuples within the partitions. At each step, the flushing policy selects a partition (say, p), among the pool of all the partitions of both the streams, with the minimum arrival probability, and flushes tuples from the partition p of the opposite stream. The RPJ switches to reactive stage whenever both the input stream blocks. In a reactive stage, the RPJ selects either an md- or dd-task based on the output generation rate of the tasks. The output generation rate of a task is determined using two parameters: the number of new results to be produced within the task, and the execution time of the task. To eliminate duplicate results during a md-task, RPJ assigns two timestamps to every tuple t: the arrival timestamp (ATS) and the departure time (DTS), where $t.ATS$ (the $t.DTS$) denotes the time that the tuple t arrived in the system (is flushed to the disk). For every partition p of stream i, the algorithm maintains a list of timestamps $T_{i,p}$ giving the invocation time of md-tasks for the respective partition. On the other hand, in a dd-task, a number of runs from the disk are merged, similar to HMJ or PMJ algorithm, to create a larger runs. To eliminate duplicate output tuples during a dd-task, the algorithm does not join two tuples from the matching runs.

Recently, a join algorithm, called RIDER [21], is proposed that processes joins using a Nested-Loop-Join algorithm. The primary objective of the algorithm is to (a) devise a flushing policy to maximize output rate, and (b) enable the system quickly switch between a in-memory stage and disk-to-disk stage. Unlike the RPJ algorithm, the RIDER algorithm does not have any disk-to-memory stage. Reference [96] proposes state spill mechanisms for a query with multiple operators. The paper points out that spilling data from one operator may affect other operators in the same pipeline. Considering this interdependency among the operators, several data spill strategies (operator level and partition level) are proposed in this paper. The state spill mechanism aims at selecting a portion of the memory state to be spilled onto disk in order to maximize the output rate. The spilled state or data are joined at the end during a clean-up phase. The PermJoin [94] algorithm spills operator states upon memory overflow, and employs a state manager that adaptively switches operators, similar to the RPJ, between joining in-memory data and disk-resident data in order to maximize overall throughput.

All these algorithms are applicable in case of joining streams with finite number of tuples, and the clean-up process occurs at the end. However, in case of the sliding window-based joins over unbounded streams, the clean up or invalidation is a continuous process and should be interleaved with the stream processing.

2.5 Join over Unbounded Data Stream

Processing joins over infinite stream usually employs a certain windowing technique to limit the memory requirements of continuous queries, thus unblocking query operators. There has been intense research works on processing stream joins over sliding windows. Here, we summarize the significant works on this issue. It should be noted that all of the algorithms shed loads during bursty stream arrivals and are not accurate.

The work presented in [79] investigates the techniques to process sliding window joins over a pair of unbounded streams. The authors introduce a unit-time-basis cost model to analyze the performance of the join techniques. The cost model consists of two separate terms each one corresponding to a join direction. Decoupling the cost of each join direction, the proposed technique demonstrates the effectiveness of an asymmetric join technique, where a different technique is used for different stream to be joined (e.g., nested loop join for one direction and hash join for the other). Although the main focus of the paper is not load shedding, it also considers the scenarios where system resources are not sufficient to keep up with the input streams.

Reference [40] considers the problem of approximate join evaluation for a pair of data streams. The paper examines the MAX-subset measure and presents an optimal offline algorithm for sliding window joins. The max-subset problem aims at maximizing *recall* of a window join operator, where the recall denotes the fraction of the result tuples produced. The optimal offline algorithm minimizes MAX-subset error in the fast CPU case under the assumption that all tuples that will arrive in future are known to the algorithm. The paper also presents some heuristics for the online join processing. The paper shows the benefit of the semantic load shedding compared to the random load shedding. Semantic load shedding adapts to resource shortages by dropping tuples based on their values, maintaining simple stream statistics. In [110], the authors consider a different model by incorporating an *importance* semantics within input tuples, and presents optimal offline algorithms and online heuristics that seek to maximize the importance of the approximate join result.

Golab et al. [61] presented and evaluated various algorithms for processing multi-way joins over sliding windows residing in main memory. Here, multi-joins are evaluated together in a series of nested for-loops, and newly arrived tuples from each window are processed separately (possibly using different join orders). Based on a per unit-time cost model, the authors propose join ordering heuristics that try to lower the number of intermediate result tuples passed down to the inner loops.

Srivastava et al. [125] propose algorithms for performing memory-limited stream joins. The age-based model for stream arrivals introduced in the paper is shown to be more appropriate, for a broad class of applications, than existing frequency-based model. In the frequency-based model, each join-attribute value has roughly a fixed frequency of occurrence on each stream. Whereas, in the age-based model, the expected join multiplicity of a tuples depends on the time elapsed since its arrival rather than on its join-attribute value. Based on this observation, the proposed algorithm conserves memory by keeping an incoming tuple in a window up to the point in time until the average rate of output tuples generated using this tuple reaches its maximum value. To help load shedding, the algorithm should know the age curve for each stream window. The age curve shows how likely a tuple within a window is to produce join tuples as it becomes older. Different applications may have different shapes (i.e., increasing, decreasing, bell-shape etc.) of the age-curves. The paper presents the hardness of the max-subset problem for arbitrary streams. Given a fixed amount memory, no online strategy can be k-competitive for any k that is independent of the length of the input sequence. Reference [6] considers CPU limitations while processing windowed stream joins, and presents a load shedding technique based on a frequency-based model. In [144], the authors consider the problem of joining streams with limited memory, and reduces the join problem to a caching one. Exploiting the known statistical properties of the input streams, the paper develops techniques to make optimal cache replacement decision based on an expected cumulative benefit (ECB) function. The ECB for a tuple indicates how desirable it is to cache the tuple. A cache replacement decision based on the ECB dominance tests leads to provably optimal algorithms for streams with known statistical properties (e.g., stream arrival rate distribution, join-attribute distribution).

In [52], the authors propose an adaptive CPU load shedding approach for two-way windowed stream joins, under CPU limitations. Instead of processing the whole window for each incoming tuple, this approach selectively processes a subset of the join window through three types of run time adaptations: adaptation to input stream rates, adaptation to time correlation between the streams, and adaptation to join directions. This approach (1) captures time based correlation between streams by maintaining statistics at segment level, and then (2) revises the selective processing decisions based on the statistics. The main idea is to prioritize the window segments (basic windows) in order to process a fraction of the window that generates higher output (time correlation adaptation), and to start load shedding from one of the windows if one direction of processing joins produces more output than the other (join direction adaptation). The join direction adaptation is carried out after the rate adaption that decides how much load the algorithm should shed. Considering the exponential explosion of possible m-way join sequences involving window segments in m streams, reference [53] extends the above algorithm for m-way windowed stream join.

As observed above, existing algorithms shed loads (e.g., drop tuples) and maximizes output results or *recall* based on a fixed or predefined model of stream arrivals (e.g., frequency-based, age-based or stochastic model). Such an approach fails to ensure exact results or results with a user defined QoS. In such a scenario, a new approach is necessary that will achieve the following goals: (a) smooth the load incorporating a secondary storage, (b) maximize output rate forgoing any predefined model of stream arrivals (i.e., placement of window data should be revised with passage of time), (c) reduce disk I/O overhead by resorting to disk-I/O amortization and by transforming the disk access patterns largely sequential.

2.6 Hybrid Join Processing

This section presents a broad discussion of the research works on join processing and real time or active data warehousing that are relevant to hybrid joins.

The issue of active or real-time data warehousing has been introduced in [22, 140, 80]. Research efforts on ETL result in the development of algorithms for specific tasks; for example, the detection of duplicates, the resumption from failure and the incremental loading of the warehouse [87, 88, 89]. These algorithms operate in a batch or off-line fashion, and thus are different from the settings considered in this paper. The work on refreshing materialized views [68, 148, 151] is relevant, but orthogonal to the problem considered in this paper. The main concern in maintaining materialized view is to decide whether the view can be maintained based on a small set of updates.

Processing joins over infinite streams, as described earlier, usually employs a certain windowing technique (time- or tuple-based) to limit the memory requirements of continuous queries, thus unblocking the query operators. Hybrid join may or may not have any window operator over the update stream; the primary assumption is that the disk-based relation is very large compared to the available memory.

Works on the joins over bounded or finite streams consider finite relations streamed over an unstable network. These join techniques access the streaming inputs and keep the received tuples in memory; upon memory overflow, these algorithms flush a subset of the tuples onto the disk and process it when CPU is idle. This model is not suited for hybrid joins where access to the disk relation is predictable and buffering stream tuples would stall the update stream compromising the requirements of the online refreshment.

In [59], the authors consider the scheduling issue while updating a data warehouse collecting data streams from a number of external sources. The objective is to choose which table (or materialized views) to update next if the multiple update jobs are pending due to the arrivals of new updates. Based on the notion of data staleness, the paper prioritize update tasks in a way to minimize average staleness. Ordering or scheduling multiple update tasks is orthogonal to the hybrid joins that corresponds to the core operation within a particular update task.

MESHJOIN [115] is the first algorithm that approaches the hybrid join. This algorithm attempts to increase the I/O efficiency by sequentially scanning the disk relation and amortizing a disk access over a large number of update tuples. MESHJOIN maintains a memory queue, that contains stream tuples arriving from the source, and a read buffer, that stores the incoming blocks from the disk. The memory queue and the disk relation are divided into same number of segments; however, the sizes of a queue-segment and a disk-segment are not the same. The algorithm reads a disk segment whenever a queue-segment is full with the incoming stream tuples; the disk segment is joined with all the segments in the memory queue. At equilibrium state, every disk invocation results in the complete processing (thus, expiration) of a queue segment. This algorithm suffers from large delays as observed in the output tuples. The primary objective of the partition-based algorithm proposed in this thesis is to reduce response time, processing overhead and disk-I/O overhead.

2.7 Distributed Processing of Stream Join Queries

Distributed processing of continuous queries of data stream attracted much research attention in recent years. Existing relevant works on diffusing loads of a stream join operator can be classified into two categories: recent advancements in continuous query and stream processing systems, and earlier work in parallel query processing.

Exploiting inexpensive shared-nothing clusters to ensure scalability without sacrificing result accuracy has not been considered in STREAM [104] or the Aurora [25] project as outlined earlier in this chapter. Reference [15] proposes a contract-based load management framework migrating workload among processing nodes based on predefined contracts. The Borealis project proposes a dynamic inter-operator load distribution mechanism by utilizing the operators' load variance coefficients [145]. In [33], the authors present architectural issues facing the design of a large scale distributed stream processing systems and outline techniques for dynamic query reconfiguration and load sharing while achieving high availability.

The Flux operator [122] extends the exchange operator [64] to support adaptive dataflow partitioning and load balancing while processing stateful operators (e.g., joins, grouping operators) over a shared-nothing cluster. The Flux operator consists of two types of intermediate operators called Flux-Prod and Flux-Cons. The Flux-Prod operator stores stream tuples (from the sources) into a buffer, and distributes the stream tuples among a number of Flux-Cons operators, that are instantiated in each of the nodes processing the stream queries. The Flux operator provides a framework for adaptively partitioning dataflows over a number of nodes. The state movement protocol within a Flux operator moves a partition from one machine to another. Within this operator, all the participating nodes maintain persistent connections with every other nodes, and the controller adjusts loads observing the CPU and memory utilization of the nodes, the sizes of partitions, and residency states (i.e., in memory or disk) of the partitions at each node.

In reference [67], the authors first address the issue of intra-operator parallelism while processing a join operator over a number of servers. The paper provides two tuple routing strategies, namely *aligned tuple routing* (ATR) and *coordinated tuple routing* (CTR), that preserve join accuracy. The ATR assigns a segment of the master stream to a selected node, and changes the assigned node at the end of every segment. All other streams align their tuple routing with the master stream. On the other hand, CTR distributes the stream segments across the participating nodes, and maintains a routing path for each stream. The routing path is a sequence of routing hop, and each routing hop V_i is a collection of nodes storing a superset of the i-th stream-window in the join order. CTR groups tuples into segments and places the segments at different nodes. A routing table records the placement of various segments within the nodes. CTR maintains optimal routing path, and incoming tuples (from a stream source) or intermediate results (generated by a segment of the intermediate *routing hop*) are forwarded, in a cascading fashion, to every node in the successive *routing hop*. The goal of the optimal routing path selection is to produce join results with minimum overhead, and is formulated into the weighted minimum set cover problem [34].

Reference [76] presents a stream database system that provides a generic framework for describing distributed execution strategies as high-level dataflow distribution templates. The query operators are specified as declarative *stream query functions* (SQFs), defined over stream units called *logical windows*. For scalable execution of queries with expensive SQFs, the paper provide a generic template for customizable data partitioning parallelism; the template consists of three phases: partition, compute, and combine. In the partition phase the window is split into sub-windows, in the compute phase an SQF is applied in parallel on each of the sub-windows, and in the combine phase the results of the computations are combined into one window. The paper implements two partitioning strategies: *window split* and *window distribute*. Each window partitioning strategy provide a partitioning template (OS-Split or S-Distribute) and a combining SQF (OS-Join or S-Merge). The partitioning schemas, which are content-insensitive, are chosen to meet scientific application requirements. Thus the issue of load imbalance across the partitions doesn't arise; For example, consider the task of computing FFT for vector of size N by computing FFT on two sub-vectors of size $N/2$ formed from the original vector by grouping the odd and even index positions, respectively. In such content-insensitive partitioning, the workload of the each SQF (i.e., FFT) over a sub-window is similar.Moreover, the paper considers a homogeneous cluster environment without any non-query background load. Thus the issues like dynamic dataflow partitioning and state movement are irrelevant to such a system.

Early works on parallel query processing concentrated on parallelizing individual join operators [120, 75]. Extensive research has been done on handling data skew (i.e., a non-uniform distribution of join-attribute values) while parallelizing an operator in a shared-nothing system (e.g., [142, 74, 83, 81, 51, 41, 71, 149, 17, 147]). These algorithms split the persistent relations in a distribution or splitting phase, and balances the loads by properly assigning the partitions or buckets across the nodes. Such a holistic approach based on the complete knowledge of data distribution in static data sets is infeasible in the streaming scenario. In [41] and [117], the authors describe how to leverage current CPU utilization, memory usage, and I/O load to select processing nodes and determine degree of declustering for hash joins. Reference [118] presents a method of parallel query processing that uses non-dedicated, heterogeneous computers. This approach doesn't partition data across the nodes; the data reside in shared stored system (e.g., Storage Area Network [56]) from which all nodes access data as needed. All the previous schemes repartitions a hash join at one point in time (i.e., between the *build* and *probe* phases) , and none of the schemes consider continual, on-the-fly repartitioning of the join operator during execution.

In addition to the work stated above, several network-aware techniques have been developed to process joins on streams with sources distributed in the network. The query operators process data within the network as the data is routed from the sources, optimizing network utilization [113, 138, 3]. Reference [86] proposes a technique to process approximate results for window joins on distributed streams (with geographically distributed source nodes) that uses Discrete Fourier Transforms (DFT), and reduces communication overheads exploiting correlation among the streams.

We consider the issue of parallelizing a sliding window join over a shared nothing cluster. As discussed in this subsection, the only existing approach to intra-operator parallelism tries to adapt a segment length to achieve the best tradeoff between join processing granularity and replication overhead, and the algorithm achieves poor load balancing. Contrary to the existing one, our proposed algorithm achieves fine-grained, intra-operator parallesim over a large number of independent nodes; the algorithm adapts loads across the nodes during execution time and reduces both communication and processing overheads.

Chapter 3

Processing Exact Window Joins

This chapter presents a brute force algorithm for processing accurate results of a sliding window joins over data streams with limited memory. While processing a number of state-based continuous queries over high rate data streams, the memory available to a DSMS may not be sufficient to store the states of the all the queries. In such cases, the secondary storages should be used to spill states of the continuous queries. Instead of shedding loads, the Exact Window Join (EWJ) presented in this chapter smooths loads by incorporating a disk storage.

3.1 Introduction

Join operators over two infinite streams are resource intensive as they need to store unbounded states for the two input streams. To limit the state size, the semantics of a join operator in a DSMS are modified to reduce the scope of the join within a window of the most recent tuples. Such a join is termed as a *sliding window join* as the window slides over the input streams with the passage of time. Sliding windows, that consist of the most recent tuples in the streams, can be either *time-based* (e.g., tuples arriving in the last 10 seconds) or *count-based* (e.g., last 100 tuples).

The real time query processing over online data streams with high arrival rates imposes a stringent limit on the processing delay of the arriving tuples (i.e., in an ideal scenario, the processing time for a tuple should not exceed the inter-arrival time of the tuples). Moreover, continuous queries with large operator states might consume a large memory during their execution. In a DSMS, where a number of continuous queries might run concurrently in the system, the main memory available for processing a query might be limited. As the stream arrival rate is bursty and unpredictable, provisioning fixed portion of memory to a join operator to handle the peak or predefined stream rate is not memory-efficient. Moreover, in some stream applications (e.g., retail management) [143] the size of the sliding window for a query might be too large to fit into memory, and secondary storage should be used to store the window data. Thus, computing state-oriented sliding window join operators may end up in memory shortages, and new techniques are necessary in executing those operators.

One promising approach to deal with the memory limitations in a graceful way is to compute the approximate results based on load shedding [53, 125, 40, 130]. However, shedding load is not feasible for queries with large states (e.g., join with large window size) as the accuracy of the query output might be below the QoS [25] specified by the user. This situation can easily arise in a scenario where the state size is far larger than the allocated memory and the majority of the tuples, including the productive ones, should be dropped. It is formally shown in reference [125] that given a limited memory for a sliding window join, no online strategy based on load-shedding can be k-competitive for any k that is independent of the sequence of tuples within the streams. Thus, for a system with QoS-based query output, secondary storage is necessary even to guarantee QoS above a certain limit. Moreover, there exist many application scenarios where continuous sliding window join queries in the system should produce exact results even though the query system may not have enough memory to cope up with the query workload at runtime [40, 96]. For example, applications like decision support, intelligence or disaster monitoring may run the risk of “missing the needle in the haystack” when provisioned with load shedding technique [40].

In such a scenario, the query processing systems should operate in cooperation with a disk archive to address the problem of a run-time memory shortage while generating complete result sets of a stream join. Instead of dropping the stream tuples during high stream load, the query system pushes the operator states or stream tuples to the disk. Such an approach has already been adopted in several research works [133, 102, 135] that aim at providing high output rate and complete query results for a single operator query over finite data sets. Reference [96] proposes mechanisms to push states onto disks for queries with multiple operators. However, in all of the above strategies, that process finite data sets, the clean-up phase occurs at the end when there does not exist any stream tuples for processing. Contrary to this scenario, in the case of a sliding window stream join with bursty input arrival patterns, the in-memory execution and disk clean-up phases should be interleaved, and thus be scheduled properly.

Algorithms for processing joins over finite streams [133, 129, 102] also suffer from the problem as stated above. Moreover, these algorithms are not I/O-efficient and do not consider *disk I/O-amortization* over large number of input tuples: operations involving flushing memory tuples and joining disk resident tuples are invoked at a partition level, and such operations require a large number of small random disk I/Os. Also, these algorithms result in a low memory utilization and a high overhead of eliminating duplicate tuples in join output (section 5.2). Thus exact processing of sliding window stream joins within a memory limited environment is a significant and non-trivial research issue as promulgated in [40, 96, 125].

Like any online processing environment [72, 129, 125], maximizing the output generation rate is a key issue in processing stream joins. The recent work [125] on processing sliding window joins selectively keeps tuples in memory aiming at maximizing the recall (i.e., total number of output tuples). This decision about placement of tuples is based on the predefined models of stream arrivals (age based or frequency based). In a data stream with time varying arrival rates of the tuples, such models might fail to capture the temporal distribution of future tuple arrivals. Hence, a generalized approach is necessary.

In this section, we consider the issue of processing exact results for sliding window joins over data streams. Using disk storage as an archive, we propose an Exact Window Join (EWJ) algorithm that endeavors to smooth the load by spilling a portion of both the window blocks and the incoming tuples onto the disk. We propose a framework for processing disk-based sliding window joins. The proposed algorithm merges the disk resident data periodically with the in-memory blocks during a phase called *disk probing*. To increase the output generation rate, EWJ algorithm employs a generalized framework to manage the memory blocks forgoing any assumption about the models (unlike previous works e.g. [125]) of stream arrival. In summary, the key contributions of the paper are as follows:

1. We propose a disk-based join algorithm EWJ that, given a limited memory, processes sliding window joins over arbitrary sequences of streaming data. The proposed algorithm produces correct output by spilling data onto the disk and retrieving/joining the spilled data within the time bound set by the window.

2. We propose a disk probing policy that, at one hand, maximizes disk efficiency by amortizing a disk scan over a number of tuples. On the other hand, the same policy maintains the productive blocks within memory, thus maximizing output rate. The proposed technique can cope with arbitrary patterns of stream arrivals.

3. We present experimental evaluation showing the effectiveness of our techniques.

The rest of the chapter is organized as follows. Section 3.2 provides the basic concept in processing sliding window queries and defines the problem. Section 3.3 provides an overview of the proposed algorithm. Section 3.4 describes the techniques to manage memory blocks, the function to integrate periodically the disk and memory blocks (disk probing), and finally

```
1: When a new tuple $s$ arrives in stream $s_i$
2:     Invalidate: Discard expired tuples in $S_{\bar{i}}[W_{\bar{i}}]$
3:     Probe: Generate $s \bowtie S_{\bar{i}}[W_{\bar{i}}]$
4:     Insert: Add tuple $s$ to $S_i[W_i]$
```

Figure 3.1: Processing a newly arrived tuple in Sliding Window Joins

the EWJ algorithm. Section 3.5 presents the experimental results. Section 3.6 concludes the paper and outlines the potential future works.

3.2 Preliminaries and Problem Definition

We briefly describe the basic model of processing continuous, sliding window queries over data streams. *Sliding windows* [8] are used to limit the state size in a stateful operator. For a stream $S_i, i = 1, 2$, we use r_i to denote the average arrival rate in stream S_i. In a dynamic stream environment, this arrival rate can change over time. Each tuple $s \in S_i$ has a timestamp $s.t$ identifying the arrival time at the system. As in [11], we assume that the tuples with a stream have a global ordering based on the system's clock. We use $S[W_i]$ to denote a sliding window on the stream S_i. Abusing the notation a little, we use W_i to denote the sliding window, where $|W_i|$ is the window size in time units. At any time t, a tuple s belongs to $S_i[W_i]$ if s has arrived on S_i within the interval $[t - |W_i|, t]$. An arriving tuple s in stream S_i is processed as shown in Figure 3.1 that shows the steps to be followed (for notational convenience $S_{\bar{i}}$ is denoted to be the opposite stream).

The basic join operator considered in this paper is a sliding window equi-join between two streams s_1 and s_2 over a common attribute A, denoted as $S_1[W_1] \bowtie S_2[W_2]$. The output of the join consists of all pairs of tuples $s_1 \in S_1, s_2 \in S_2$ such that $s_1.A = s_2.A$, and $s_1 \in S_1[W_1]$ at time $s_2.t$ (i.e., $s_1.t \in [s_2.t - |W_1|, s_2.t]$) or $s_2 \in S_2[W_2]$ at time $s_1.t$ (i.e., $s_2.t \in [s_1.t - |W_2|, s_1.t]$. We assume that the amount of memory available for storing the join operator states or windows is less than the size of the windows. Hence, a portion of each of the stream window W_i resides on the disk. Figure 3.2 illustrates the join processing architecture. The join operator, that has buffers attached with its input streams, fetches tuples from the input buffers, processes the join, and sends the output as a stream. The join algorithm should be I/O efficient and minimize per tuple disk I/O time by amortizing a disk access over a large number of tuples from the opposite stream.

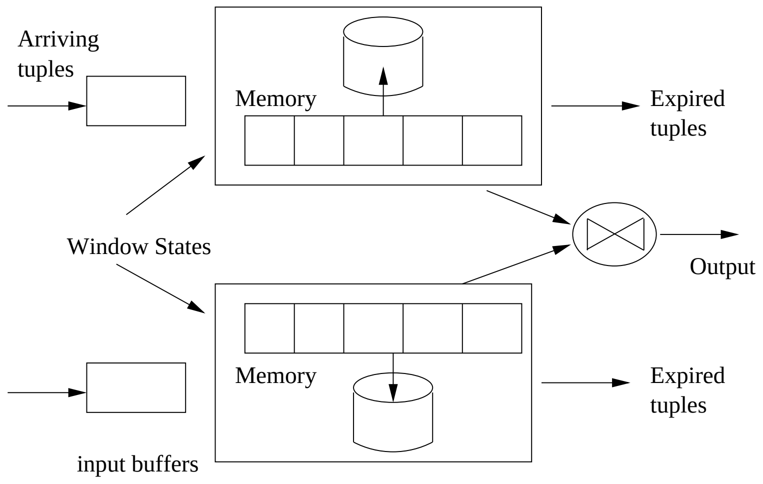


Figure 3.2: Architecture for processing exact, sliding window stream joins

3.3 Solution Overview

We propose an algorithm called *Exact Window Join* (EWJ) that computes sliding window joins between two streams without compromising the output accuracy. Existing algorithms shed the load by dropping tuples selectively. Contrary to these algorithms, we attempt to defer the load during high workload, and process the deferred load during the period of low workload. Thus, EWJ aims at smoothing the load using the disk as secondary storage.

The proposed algorithm EWJ is based on the framework given in Figure 3.3 showing the organization of tuples within a stream window W_i. We do not assume any hashing and partitioning scheme that divides the incoming stream into multiple partitions and joins the individual partitions locally. Thus we assume a single partition, and any partitioning scheme can use the proposed algorithm within each partition. For each stream $S_i(i = 1, 2)$, EWJ stores the relevant tuples (window states) in the memory and on the disk. We denote the portion of the window $S_i[W_i]$ residing in memory and disk as W_i^{mem} and W_i^{disk} respectively. We assume that the sizes of a memory page and a disk block are equal, and use the terms interchangeably. We arrange the tuples into pages (or blocks) and process the tuples at the granularity of a page or block. In addition to the timestamp of the tuples within a page, each page or block contains a unique sequence number (or timestamp). All tuples within a block are homogeneous in the sense that they are probed against the same number of blocks from the window of the opposite stream $(S_{\bar{i}}[W_{\bar{i}}])$.

The memory segment of a window W_i is divided into an *invalidation segment* and a *generative segment*. An invalidation segment is allocated to make the tuple expiration efficient, whereas the generative segment stores the more productive blocks to maximize the output

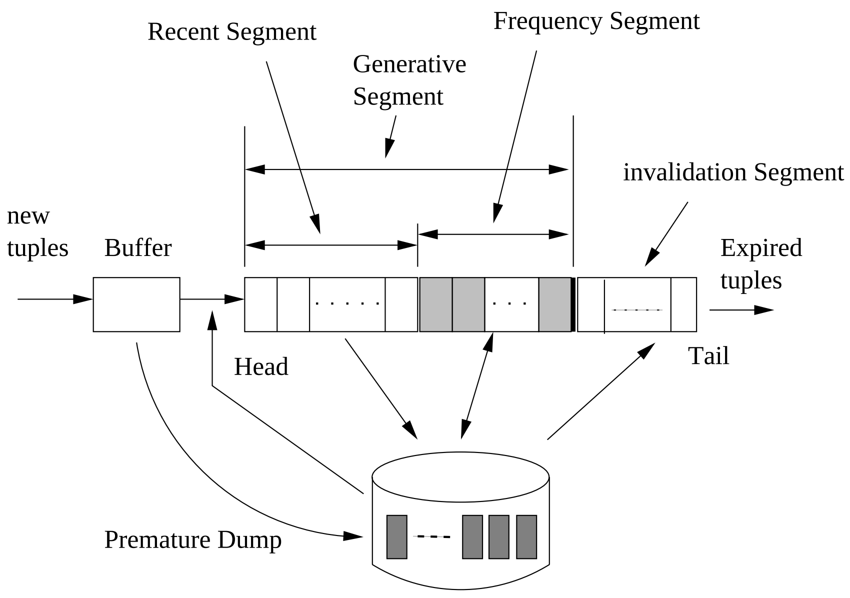


Figure 3.3: Organization of incoming tuples within a window: In memory, tuples are divided into generative and invalidation segments. A generative segment is divided into Frequent and Recent Segments. Incoming tuples are are dumped onto the disk on buffer overflow, and also after *disk probing*.

throughput. We explain the rationale and management of the segments in the subsequent sections. An incoming tuple s in stream S_i is joined with the tuples in the invalidation and generative segments (within $W_{\bar{i}}^{mem}$). Incoming tuples are packed into blocks and dumped periodically onto the disk. both from the buffer (when the arrival rates exceed the processing capacity) and from the generative segment. We use the terms *premature dump* and *mature dump* to refer to the former and the later dumping activities, respectively. In case of the mature dump, a block (from the generative segment of W_i) is dumped onto the disk only when it is joined with all the blocks within the disk portion of the window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{disk}$) at that time. This eliminates the disk-disk phase of joins as in [129, 102]. Mature dump requires periodic scanning of the disk to retrieve the blocks from $W_{\bar{i}}^{disk}$ and joining with the pages to be dumped. Mature dump occurs when the join memory is full and a portion of the memory should be made free. We present the details of the proactive processing in the subsequent section.

Premature dump occurs when the input buffer overflows and is handled by the buffering algorithm. We implement the buffer using a simple algorithm similar to one proposed in [125]. We are not concerned about the efficiency of the buffering mechanism as we do not take the buffering delay into our consideration. We are only concerned with the effectiveness of the join operator. For our system, we try to process a sequence of tuples (with timestamps) within the time restriction of the window. Here, the size of the buffers provides an indication regarding the processing rate of the join operator: if the system is supplied with a load exceeding its processing capacity, the buffer size increases without any bound rendering the system unstable.

3.4 Exact Join

In this section, we describe in details the join algorithm for processing sliding-window join for streaming data. The algorithm consists of three major sub-tasks: invalidation of the tuples, maintaining the generative segments, and probing the disk periodically to join the disk resident data with the incoming data. Before presenting the algorithm we illustrate the key ideas encompassing the three majors subtasks. The generative segment of a window W_i is divided into two segments: Recent segment (W_i^{rec}) and Frequent segment (W_i^{freq}). So, the memory portion of each window W_i has three segments in total: invalidation segment (W_i^{inv}), Recent segment and Frequent segment.

3.4.1 Invalidation

As the window slides over time, tuples from the tail of the window should be discarded continuously. To reduce the disk access time in eliminating tuples, we reserve a few blocks in memory to store the tail of the window and term the blocks as *invalidation segment*. Without this invalidation segment (W_i^{inv}), in the worst case, we need to access the disk for each incoming tuple. Using this invalidation segment, we discard the tuples from the memory blocks whenever the window slides over time.

3.4.2 Managing Generative Segments

The purpose of using the generative segments is to maximize the output throughput of the join between the pair of data streams. This can be achieved by keeping the more productive blocks in memory. Here, productivity of a block at a particular time refers to the number of output tuples generated by the block joining with the tuples from the opposite stream. As observed in [125], different tuples (or blocks) exhibit different characteristics (i.e., age-based or frequency based) while joining with the incoming tuples or blocks of the opposite stream. To capture both the temporal and frequency phenomenon in joins, we divide the generative segment into *Recent Segment* and *Frequent Segment*. The *Recent Segment* contains the arriving blocks sequentially, while the *Frequent segment* contains the blocks that show long term frequency of output tuple generation. The incoming blocks (in stream S_i) are added to the Recent segment at one end, while the blocks from the other end are joined with the disk resident blocks from the opposite window ($W_{\bar{i}}^{disk}$) and stored to the disk in mature dump. On the other hand, blocks in Frequent segment are stored in the memory as long as the output generation rate corresponding to these blocks are high. Blocks contained in the Frequent segment also have their disk counterpart, but the disk image of these blocks are skipped while scanning the disk.

Managing blocks in a Recent segment

The recent segment contains blocks arriving during an interval in the recent past. When the segment is close to overflow, the blocks from this segment are dumped onto the disk. In our join algorithm, we maintain the following constraint:

> Constraint **a**: *At a particular time, the blocks from the Recent segment of stream S_i should join with all the blocks in the disk portion of window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{disk}$) at that time before being dumped to the disk (during mature dump)*

Blocks in a Recent segment are dumped to the disk periodically using *disk probing* phase as described in subsection 3.4.3. Disk probing phase starts when a fraction r of Recent segment is filled with the newly arrived blocks. During this phase, blocks retrieved from the disk are joined with all the blocks in the Recent segment, and the blocks in the Recent segment are dumped onto the disk. However, we do not discard the blocks from the Recent segment immediately after scanning the disk. Such removal of the blocks has the drawback of reducing the output generation rate after the mature dump as an incoming tuple in the opposite stream finds the Recent segment empty after the dump. Hence, the memory of the Recent segment is not properly utilized, and the Recent segment might fluctuate between overflow and empty states: on the average half of the Recent segment remains unutilized. Thus, we discard the blocks, that are already joined with the disk portion of the opposite window, from the recent region only when incoming tuples demand a new block. This way we amortize a disk scan over almost all the blocks in the Recent segment, and at the same time maximize the output throughput.

Managing blocks in a Frequent segment

Blocks displaying a good long term frequency distribution in generating output tuples are stored in this segment. Each block has a field called *productivity* that stores the number of output tuples generated by the tuples within the block. The decision about placement or replacement of a block in Frequent segment is based on its productivity value. This decision is made periodically after a certain interval. The productivity values of the blocks are reset during this update stage. So, all the blocks within the window should be accessed during this update step. The feasible time to carry out this update step is during the mature dump when the disk blocks are scanned.

During the update stage, as the disk blocks are scanned, a block is brought into the Frequent segment if its productivity exceeds that of a block already in Frequent segment by a threshold (δ). The block having the minimum productivity value among the blocks in the Frequent segment is replaced. A block b_i^p evicted from the Frequent segment of window W_i is already joined with the blocks in the Recent segment of the stream $W_{\bar{i}}$. If that block is not already scanned on the disk, it will be read and be joined with the blocks in the Recent segment of window $W_{\bar{i}}$. To prevent this duplicate processing, we maintain a list E_i containing the block numbers of evicted blocks. If an incoming block from the disk is contained in E_i, we omit processing that block. List E_i is reset to *null* at the end of the update stage. The productivity values of the blocks in Frequent segment are reset after the update stage.

3.4.3 Disk Probing

After introducing above the basic concepts behind the memory management in join processing, we describe the disk probing operation. First, we introduce the technique to keep necessary information at the block level to avoid duplicate output tuples and to ensure the completeness of the generated output. Then, we present the probing algorithm that scans the disk blocks and joins the scanned blocks with the blocks in the Recent segment of the joining window.

As described earlier, blocks from the Recent segment are removed on demand. Such passive removal of the blocks might lead to duplicate output generation. A block b_i^j in the Recent segment of window W_i may join with a block $b_{\bar{i}}^k$ in the Recent segment of stream $S_{\bar{i}}$. Later b_i^j participates in mature dump and is stored on the disk. However, due to the passive removal of the blocks from the Recent segment, the same block remains in memory. Now, when block $b_{\bar{i}}^k$ of the opposite window participates in mature dump it finds the block b_i^j, already joined in previous step, on the disk while scanning. We use the sequence number of a block, denoted as *block number*, in solving this issue: every incoming block is assigned an unique number from an increasing sequence. Every block b_i^k in Recent segment of S_i stores the minimum block number (*minBN*) among the blocks, from the Recent segment of the opposite window, that b_k^i joins with. When b_i^k participates in mature dump, it joins with a block $b_{\bar{i}}^p$ if the block number

Notations	Description
S_i, $S_{\bar{i}}$	stream i and opposite to i, respectively
W_i, $W_{\bar{i}}$	Window of stream S_i and $S_{\bar{i}}$, respectively
W_i^{disk}	Disk portion of window W_i
W_i^{rec}	recent segment of window W_i
W_i^{freq}	Frequent segment of window W_i
W_i^{inv}	invalidation segment of window W_i
W_i^{mem}	memory portion of window W_i, ($W_i^{rec} + W_i^{freq} + W_i^{inv}$)
$W_i^{rec}[H]$	block at the head of W_i^{rec}
b_i^p	block p in window W_i
$b_i^p.minBN$	minimum block number from $W_{\bar{i}}^{disk}$ & $W_{\bar{i}}^{rec}$ that $b_i^p \in W_i^{rec}$ joins with
$b_i^p.prod$	Productivity of block b_i^p

Table 3.1: Notations and the system parameters (EWJ Algorithm)

of the block $b_{\bar{i}}^p$ is less than the ***minBN*** value of the block b_i^k, i.e., $b_{\bar{i}}^p.BN < b_i^k.minBN$. As b_i^k is already in W_i^{mem}, any block in $W_{\bar{i}}$ arriving after $b_i^k.minBN$ is already joined with b_i^k. So, during disk probing any block $b_{\bar{i}}^p \in W_{\bar{i}}^{disk}$ having $b_{\bar{i}}^p.BN \geq b_i^k.minBN$ can be omitted.

On the other hand, we observe that any block b_i^p in Frequent segment of window W_i (i.e., W_i^{freq}) is already joined with all the blocks in window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{rec}$ and $W_{\bar{i}}^{disk}$). When a block is brought into the Frequent segment, it is already joined with the blocks in disk portion of window $W_{\bar{i}}$, i.e, $W_{\bar{i}}^{disk}$ (refer to **Constraint a** stated earlier), and that block is joined with $W_{\bar{i}}^{rec}$(Recent segment of the opposite window) immediately before being placed on the Frequent segment. Now, any incoming block in window $W_{\bar{i}}$ is joined with b_i^p. Moreover, if any block is dumped on the disk, it has already found b_i^p on memory and has been joined with that block. So, blocks in the Frequent segment need not store any extra information, and no issue of duplicate processing other than the one mentioned in section 3.4.2 arises.

All the blocks within a window need to store the productivity values. Updating this value for a memory resident block is easy; however, updating the productivity value for a disk resident block requires one disk write to update the corresponding block. To make this update of productivity values efficient we maintain the productivity values of all the disk resident blocks in memory as a list of tuples <*block number, productivity*>, and use the term productivity table to refer to this list. Such a table consumes little space and might be stored in the memory. The productivity values of the blocks in a Disk segment are updated during disk probing. We enumerate the variables or notations in Table 1.

Having outlined the basic concepts, we present the disk probing algorithm. Disk probing phase starts when a certain fraction (r) of the memory allocated to the Recent segment is occupied by the blocks that have not participated in disk probing.

Algorithm 1 DISKPROBE(W_i^{rec}, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, $uFlag$)

```
Input: $W_i^{rec}$, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, $uFlag$
Output: produce output tuples; update frequent segment if $uFlag$ is True
 1: position disk head to the first block in $W_{\bar{i}}^{disk}$
 2: $E_{\bar{i}} \leftarrow null$
 3: while exists new block in $W_{\bar{i}}^{disk}$ do
 4:     read block $b_{\bar{i}}^m$ from disk
 5:     for each block $b_i^n \in W_i^{rec}$ do
 6:         if $b_{\bar{i}}^m.BN < b_i^n.minBN$ and $(b_{\bar{i}}^m \notin W_{\bar{i}}^{freq}$ or $b_{\bar{i}}^m \notin E_{\bar{i}})$ then
 7:             join $b_i^n$ with $b_{\bar{i}}^m$
 8:             update $b_{\bar{i}}^m.prod$ and $b_i^n.prod$
 9:             $b_i^n.minBN \leftarrow null$
10:             if $uFlag$=TRUE and $b_{\bar{i}}^m.prod > b_{freq(\bar{i})}^{min} + \delta$ then
11:                 Evict $b_{freq(\bar{i})}^{min}$ from $W_{\bar{i}}^{freq}$ and add to $E_{\bar{i}}$
12:                 insert block $b_{\bar{i}}^m$ into $W_{\bar{i}}^{freq}$
13:             end if
14:         end if
15:     end for
16: end while
```

Function DISKPROBE(W_i^{rec},$W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, $uFlag$), as shown in Algorithm 1, is invoked periodically when a fraction r of the Recent segment of window W_i (W_i^{rec}) is occupied with the fresh blocks—blocks that did not participate in disk probing earlier. The function scans the disk retrieving blocks in $W_{\bar{i}}^{disk}$ and joins the blocks read with those in W_i^{rec}. The blocks in Frequent segment of window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{freq}$) are updated periodically, and this update phase is carried out with disk probing. Moreover, this *update interval* might be longer than the *probing interval*—the interval between two consecutive disk probes in a stream. So, the parameter $uFlag$ is used to indicate when to update the relevant Frequent segment(i.e., $W_{\bar{i}}^{freq}$). To eliminate duplicate tuples in output, an incoming block from the disk is checked against the condition in line 6 before joining with W_i^{rec}. The first part of the condition, as described earlier in this section, ensures that the disk block was not joined with the memory resident block (from W_i^{rec}) before being dumped onto the disk, while the second part checks whether the incoming block is in Frequent segment or recently evicted from the Frequent segment(c.f., 3.4.2). In line 8, the productivity values of the blocks are updated on the productivity table avoiding expensive disk writes. Line 10–14 update the Frequent segment $W_{\bar{i}}^{freq}$.

Algorithm 2 EWJ

Input: data streams S_i,window length $W_i(i = 0, 1)$;
Output: results of stream joins

```
 1: initialize variables $freshN_i$, $UI_i$, $BN_i$, $W_i^{rec}[H].BN$ {set to 0; i=1, 2}
 2: loop
 3:     retrieve a tuple $s_i$ from input buffer of $S_i$ in FIFO order
 4:     $W_i^{rec}[H] \leftarrow \{W_i^{rec}[H] \cup s_i\}$
 5:     compute $s_i \bowtie \left\{W_{\bar{i}}^{inv} \cup W_{\bar{i}}^{rec} \cup W_{\bar{i}}^{freq}\right\}$
 6:     invalidate tuples/blocks
 7:     if $W_i^{rec}[H]$ is full then
 8:         if $W_i^{rec}$ is full then
 9:             emit the last tailing non-fresh block
10:         end if
11:         shift blocks in $W_i^{rec}$ to right
12:         $W_i^{rec}[H].BN \leftarrow BN_i$
13:         increment $BN_i$
14:         increment $freshN_i$
15:     end if
16:     if $freshN_i \geq r \times n_{rec(i)}$ then
17:         dump to disk $freshN_i$ fresh blocks from $W_i^{rec}$
18:         $UI_i \leftarrow UI_i + freshN_i \times |b_i^m|$
19:         $freshN_i \leftarrow 0$ { reset $freshN_i$ }
20:         if $UI_i \geq Epoch_i$ then
21:             DISKPROBE($W_i^{rec}$, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, 1)
22:             $UI_i \leftarrow 0$ {reset the count}
23:         else
24:             DISKPROBE($W_i^{rec}$, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, 0)
25:         end if
26:     end if
27: end loop
```

3.4.4 Join Algorithm

Having described the details of the techniques behind the join algorithm, we now present the join algorithm EWJ in Algorithm 2. Within the join algorithm an infinite loop fetches, in each iteration, tuples from an input buffer and joins the fetched tuples with the opposite stream. At each step, the stream having a pending tuple/block with lower timestamp is scheduled. Tuples fetched from the buffer is accumulated into the block at the head of the Recent segment W_i^{rec} that is organized as a list of blocks. Here, the head of the segment denotes the first block and the tail denotes the last one. The incoming tuples are then joined with the memory portion of the window $W_{\bar{i}}$ (W_i^{mem}). Line 3–6 shows the corresponding code segment. Here, the buffers within each stream are stored in both memory and disks using the read/write mechanisms as proposed in [103], and this buffering mechanism is transparent to the join algorithm.

If the block at the head of W_i^{rec} is full(line 7), all the blocks in W_i^{rec} are shifted towards the tail spilling the non-fresh block at the tail and leaving the block at the head empty. The variable $freshN_i$ tracks the number of blocks in W_i^{rec} that did not participate in a disk probing. When a fraction r of the blocks are occupied by the newly arrived tuples, the disk probe phase is invoked (Line 16). Line 17 stores the blocks to disk that participated in the disk probing. However, the blocks participating in the disk probe phase remains in the recent segment until a newly arriving block emit it from the tail(as in line 9); here, $|W_i^{rec}|$ denotes the total number of blocks within W_i^{rec}. It should be noted that, for r-value less than 1 there must always be a non-fresh block to emit in line 9.

Blocks in Frequent segment $W_{\bar{i}}^{freq}$ are updated periodically at the end of an epoch $Epoch_i$. This update is carried out during disk probing phase. The epoch can be measured either by time or by the number of tuples processed. We use the later in specifying the epoch. 'Disk Probe' with the update phase (in line 21) is invoked if the number of tuples processed (UI_i) since the last update exceeds the epoch length; otherwise, *disk probing* omits the update phase setting the last parameter to 0 (line 24).

3.4.5 Adjusting sizes of Invalidation and Recent Segment

We store the disk segment in the form of basic windows [52, 58] where blocks within a basic window are physically adjacent and can be accessed without any extra overhead. Let us consider a uniform arrival rate λ_i in a stream S_i within a time interval t. So, the total disk access time in handling the arriving tuples,

$$T_{disk}^{rec(i)} = \left\lfloor \frac{\lambda t c_{tup}}{r|W_i^{rec}|c_b} \right\rfloor \left(RT(W_{\bar{i}}^{disk}) + WT\left(r|W_i^{rec}|\right) \right) \tag{3.1}$$

Here, $RT(n)$ and $WT(n)$ refer to the total time to read and write n blocks respectively; c_b and c_{tup} denote the sizes of a block and a tuple, respectively.

Based on our storage model for the disk segment, the time to read n blocks can be written as,

$$RT(n) = \left\lceil \frac{n}{B_{win}} \right\rceil (C_L + C_S) + \frac{nc_b}{B} \tag{3.2}$$

Where,	C_L	$=$	disk latency(sec)
	C_S	$=$	seek time (sec)
	B	$=$	Disk bandwidth (mbps)
	B_{win}	$=$	size of a basic window Disk (in blocks)

On the other hand, total disk time required for refilling the invalidation segment is,

$$T_{disk}^{inv(i)} = \left\lceil \frac{\lambda t c_{tup}}{|W_i^{rec}|c_b} \right\rceil RT\left(|W_i^{inv}|\right) \tag{3.3}$$

As we consider a uniform stream arrival rate, the total number of tuples invalidated within the interval t is also equal to the total number of tuples arrived (λt). Hence, equation 3.3 provides the time to retrieve λt tuples from the disk. The sizes of the invalidation and recent segment can be periodically adjusted based on the observed disk delays within a period.

3.5 Experiments

This section describes our methodologies for evaluating the performance of the EWJ algorithm and presents experimental results demonstrating the effectiveness of the proposed algorithm. We begin with an overview of the experimental setup.

3.5.1 Simulation Environment

We evaluated the performance of the prototype implementation using synthetic traces. All the experiments are performed on an Intel 3.4 GHz machine with 1GB of memory. We implemented the prototype in Java. As a buffer, we allocated 2 MB of memory and divided this memory between the streams. We incorporated the disk as a secondary storage while buffering the arriving tuples [103]. Here, it should be noted that we have not measured the processing delay for the tuples in buffers. The main focus of our experimentation is to observe the effectiveness or the efficiency of the join processing algorithm. The number of the buffered tuples indicates when the system is overloaded or saturated; in such a scenario, the number of buffered tuples increases with time. Thus the size of the buffer is virtually unbounded and plays no role in the experiments.

Traces

We generate the time varying data streams using two existing algorithms: PQRS [137] and *b-model* [136]. PQRS algorithm models the spatio-temporal correlation in accessing disk blocks whereas the b-model captures the burstiness in time sequence data. In the experiments, we generate data streams for a certain duration T. We observe that the time to generate a large volume of data tuples within T while using the PQRS algorithm is prohibitively large as it requires us to sort the tuples based on their timestamps. We divide the duration T into 2^n mini-intervals, measure the total volume of tuples within each mini-interval using *b-model*. We choose the aggregation level (n) in *b-model* so as to bound the maximum tuple volume within a mini-interval to a given value: this upper limit varies with the parameter b in *b-model* (e.g., for b=0.6 and T = 4 hours, we set the upper limit to 70000 tuples). Having determined the total volume of tuples within a mini-interval, these tuples are generated using the PQRS algorithm. In our settings, the average arrival rate (for a stream) denotes the long term average within T, and the instantaneous arrival rates depends on the parameters capturing burstiness both in PQRS and *b-model*.

Parameter	Defaults	Comment
$\lvert W_i\rvert(i = 1, 2)$	0.5	Window length(hr)
λ	200	Avg. arrival rate(tuples/sec)
δ	100	Threshold to update freq-segment(tuples)
α	0.5	Decay parameter for the productivity value
r	0.95	Threshold to invoke disk-probe
b	0.6	burstiness in traces (captured by *b-model*)

Table 3.2: Default values used in experiments (EWJ Algorithm)

System parameters, metrics and default values

To model the access time for the disk segment, we divide a disk segment (W_i^{disk}) into n_i basic windows $B_{ij}^{win}(j = 1 \dots n_i)$ [52]. We assume that disk blocks within a basic window are physically contiguous, and the delay in accessing the blocks in a basic window is estimated only by the disk bandwidth(i.e., no seek or latency). However, accessing blocks beyond the boundary of a basic window imparts an extra delay equivalent to the seek time and rotational latency (i.e., an access time). As a base device, we consider IBM 9LZX and use its parameters in measuring the access time.

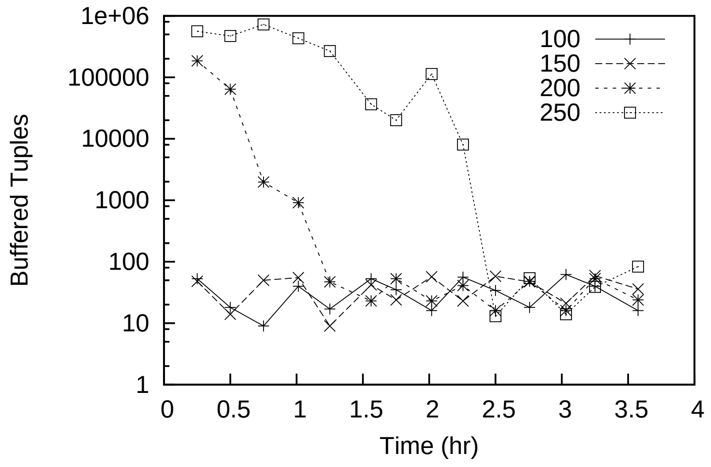


Figure 3.4: Buffer size at different time points

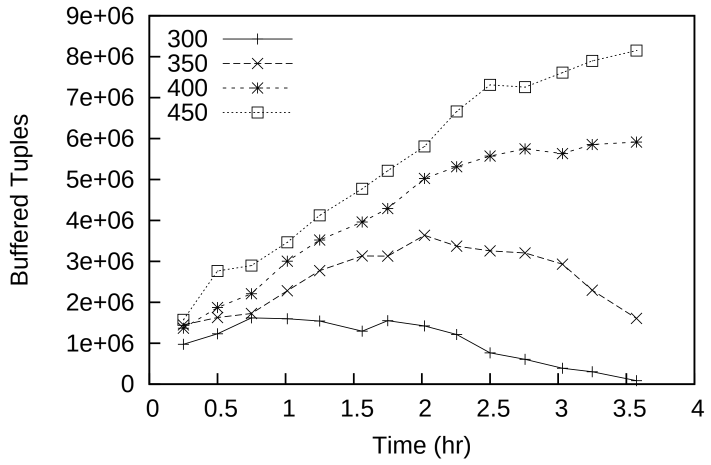


Figure 3.5: Buffer size at different time points

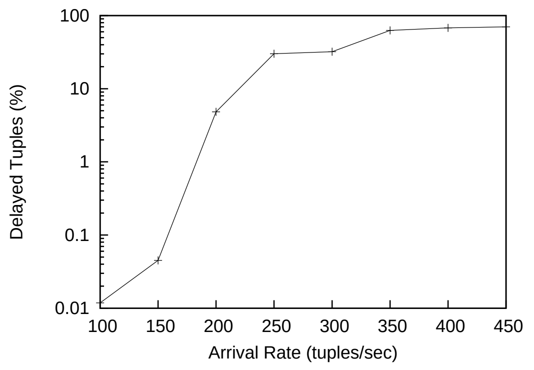
In our experiments, we fix the memory page and also the disk block size to 4KB. Each record has a length of 128 bytes. The domain of the join attribute A is taken as integers within the range $[0 \ldots 10 \times 10^6]$. We allocate 1MB of memory to buffer the stream tuples generated up to a particular instant. We divide the memory between the buffers in the two streams. When a buffer overflows, we dump the incoming tuples onto the disk. Hence, the total number of the buffered tuples indicates the degree of overload of the system. In normal cases, the buffer size or the volume of pending tuples increases within a bursty interval and decreases while the load is not very high. But, when the system is permanently overloaded, the volume of buffered tuples increases with time saturating the whole system.

In addition to the total pending or buffered tuples, we measure *average production delay*, total CPU time, total disk time and maximum size of disk segment. We measure the delay in producing an output tuple as the interval elapsed since the arrival of the joining tuple with more recent timestamp. For example, if tuples s_1 and s_2 are the joining tuples of the output tuple (s_1, s_2), where $s_1.t > s_2.t$ (s_1 being the more recent) and current time is T_{clock}, then the delay in producing the output tuple is $(T_{clock} - s_1.t)$. This metric (i.e., average production delay) indicates how quick an output tuple is generated; hence, can be considered as an indication of instantaneous output generation rate. We also measure the percentage of *delayed tuples* that indicates the fraction of tuples not processed within the time limit of the stream window. It happens due to the delay in bringing the blocks from a disk segment to the invalidation segment. Unless otherwise stated, the default values used in the experiments are as given in Table 3.2.

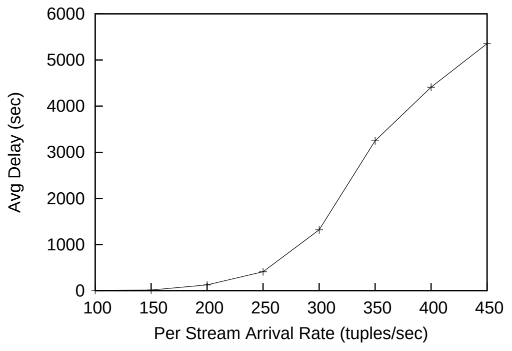
3.5.2 Experimental Results

In this section, we present a series of experimental results for assessing the effectiveness of the proposed join algorithm. We measure total number of buffered tuples from both of the joining streams, average delay in generating an output tuple, maximum length of a stream window and percent of delayed tuples. For each set of experimentation, we run the simulator for 4 simulation hours. We start to gather performance data after an startup interval of 30 minutes is elapsed.

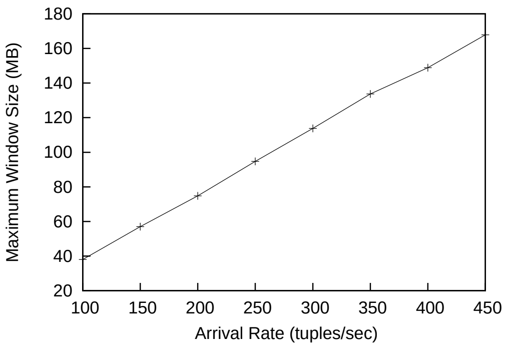
We first present the experimental results with varying stream arrival rates. Figure 3.4 and Figure 3.5 show the total buffered tuples (from both the joining streams),for varying stream arrival rates, at different time-points. As shown in the figures, the numbers of the buffered tuples remain within a bound for the arrival rates even up to 300 tuples/sec. However, for the arrival rates above 250 tuples/sec, the size of the pending tuples becomes very large. For example, for arrival rate 300 tuples/sec, the size of the pending tuples rises up to around 2



(a) Percentage of delayed tuples



(b) Average delay



(c) Maximum window size (MB)

Figure 3.6: Different metrics with varying stream arrival rates

million overloading the system.

As observed in Figure 3.5, a large fraction of the tuples cannot be processed within the window limit, and the fraction of tuples failing to be processed within the window bound increases drastically with the increase in the arrival rates beyond a certain limit. The tuples that cannot be processed within the time limit of the stream window is denoted as *delayed tuples*. Figure 3.6 shows the effect of varying arrival rates on the following three metrics: the percentage of delayed tuples, the average delay and the maximum window size. These delayed tuples get expired at a time later than the usual after they get joined with the respective window. It is observed in Figure 3.6(a) that the percentage of the delayed tuples increases sharply with the increase in arrival rates beyond 150 tuples/sec/stream. It should be noted that even though the size of the pending tuples is not high, a small fraction of the tuples misses the timing constraint for the arrival rates less than 150 tuples/sec/stream.

Figure 3.6(b) shows the average delay in producing output tuples with the increase in ar-

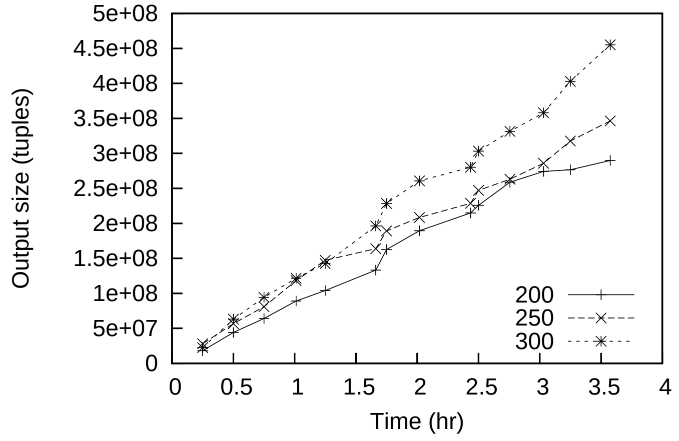


Figure 3.7: Output size(tuples) at different time points

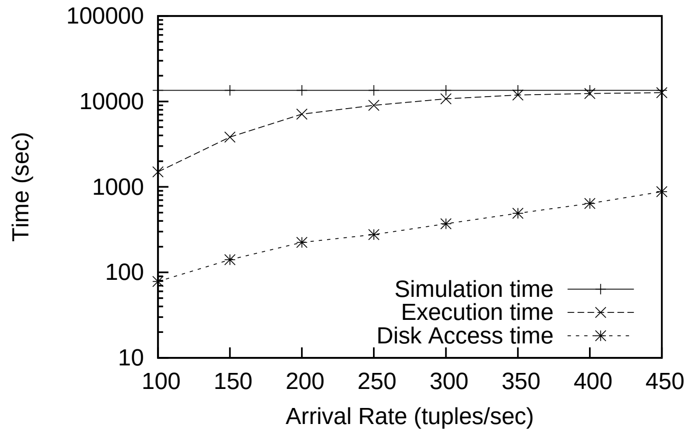


Figure 3.8: Total system times with varying stream arrival rates

rival rates. Figure 3.6(c) shows the maximum window size during the system activity. Though the allocated memory per stream window is 20MB, spilling the extra blocks onto the disk does not impart significant increase in average output delay even for the arrival rates up to 250 tuples/sec. Techniques based on load-shedding would have discarded the extra blocks beyond 20MB losing a significant amount of output tuples that would never have been generated.

The number of generated output tuples up to a particular time-point is seen on the Figure 3.7 separately for different arrival rates. As expected, the rate of output generation is higher for high arrival rates. Figure 3.8 shows the processing or execution times (CPU and disk time) for varying arrival rates; the disk time is also shown separately. Here, it is observed that there lies a wide gap between the simulation time and the execution time (note the log scale on the y-axis), and this gap vanishes as the arrival rate increases. This implies that the

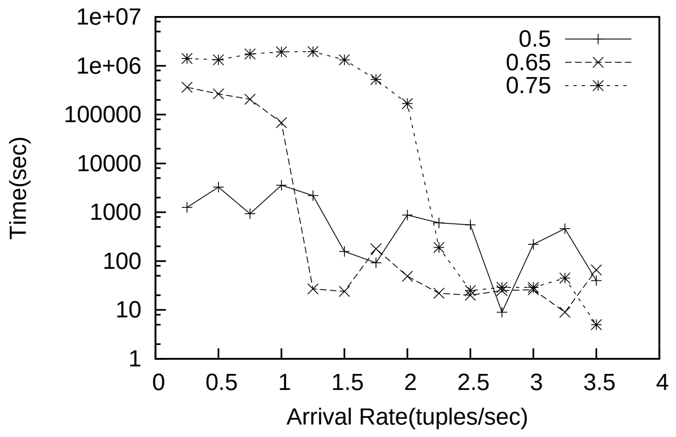


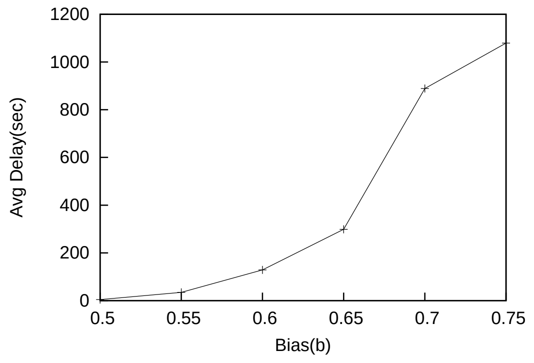
Figure 3.9: Buffer size at different time points for different biases

utilization of the resources (CPU and disk) is low for the arrival rates below 200 tuples/sec. Hence, the system remains idle even though there are a significant amount of delayed tuples (c.f., Figure 3.6(a)).

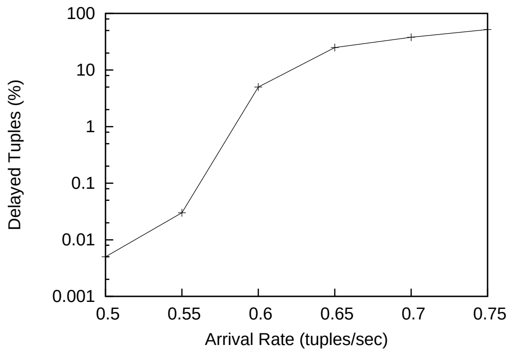
The above anomaly can be attributed to the burstiness in workload: when stream arrival rate is very low, the processor remain idle, and during bursty interval the CPU can not process the tuples within the proper time frame. Moreover, we observe that total disk access time is very low. Hence, it is obvious that the performance of the system is CPU-bound even for a decent arrival rates of 250 tuples/sec. The disk-probe that occurs only when the recent segment becomes (almost) full. Thus, during the below par system load, the CPU and disk remain idle though there lies a large amount of blocks within the recent segment to participate in disk probe. Thus, scheduling the disk probe proactively based on the recent utilization of the resources might decrease both the average production delay and the number of delayed tuples.

Figure 3.9 show the variations in the number of the buffered tuples with the variations in burstiness (bias b in the *b-model*) of the generated traces. It is shown in the figure that the number of buffered tuples oscillates with time even when b is 0.5. Such variations can be ascribed to two factors. one is the spatio-temporal correlation used in the PQRS algorithm that generates the tuples within a mini-interval (c.f.,3.5.1). And the other factor is the disk probe that occurs after a long interval and lasts for a significant duration; hence, the tuples accumulate on the buffer while *disk probe* is in progress.

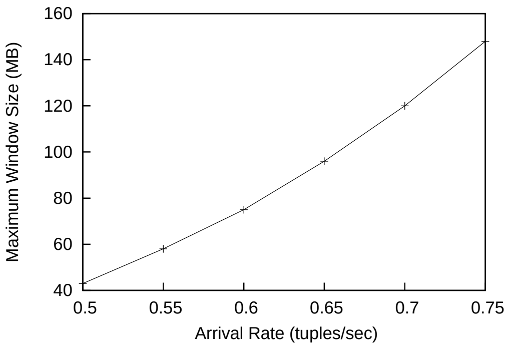
Figure 3.10 shows, for different biases (or burstiness), the variations in the following three metrics: the average delay, the percentage of delayed tuples and the maximum window size. The production delay increases drastically with the increase in burstiness (Figure 3.10(a)).



(a) Average delay



(b) Percentage of delayed tuples



(c) Maximum window size (MB)

Figure 3.10: Different metrics with varying bias (burstiness)

The variations in the percentage of *delayed tuples* and the maximum widow size with the varying burstiness are shown, respectively, in Figure 3.10(b) and Figure 3.10(c).

3.5.3 Comparison with Relevant Research

Comparison with Relevant Research: In our research, we consider the problem of producing exact results for stream joins given a limited memory. Existing approaches shed load during peak workload by dropping tuples(e.g., [40, 125, 130], and/or processing (e.g., [52]). These approaches fail to produce exact results within an environment where data arrival rate is bursty in nature and/or the length of the stream windows exceeds the available memory. Our scheme handles the burstiness in stream arrivals by spilling data on the disk upon buffer overflow; and it copes with large window by spilling window data on the disk.

The experimental results show that the generation of exact results with bounded average production delay is feasible. We observe that the total disk overhead in coping with the bursty workload is very little; thus, the performance is CPU-bound even for a window size of 0.5 hour and a stream arrival rate of 200 tuples/sec in each of the joining streams. Also, the fraction of delayed tuples processed to refine the output results remains very low (less than 0.4%) even for average arrival rate up to 200 tuples/sec/stream. Even though we do not capture the buffering delay separately, such delays are reflected in the average production delay as the timestamps are attached with the tuples before they enter the buffer.

3.6 Conclusion

In this chapter, we address the issue of processing exact, sliding window joins between data streams. We provide a framework for processing the exact results of an stream join, and propose an algorithm to process the join in a memory limited environment having burstiness in stream arrivals. Storing the stream windows entirely in memory is infeasible in such an environment. To maximize output rates, we propose a generalized framework to keep highly productive blocks in memory and to maintain the blocks in memory during systems activity, forgoing any specific model of stream arrival (e.g., age based or frequency based model [125]). When the window size is large compared to available memory, EWJ spills blocks onto the disk and integrates periodically with the tuples on the recent segment of the opposite stream. We present the experimental results to demonstrate the effectiveness of the algorithm. We observe that the performance of the join processing algorithm with a large window becomes CPU-bound for a per-stream arrival rate around 250 tuples/sec.

Chapter 4

Rate-based Progressive Window Joins

In this chapter, we present a methodology to process window-based stream joins called Rate-based Progressive Window Join (RPWJ). The RPWJ is based on an existing algorithm Rate-based Progressive Join (RPJ) [129]—a variant of XJoin [133]. RPWJ incorporates substantial design changes. The temporal aspects of data within a sliding window invalidates the application of sort-merge join during a reactive stage. Therefore, the estimation of the output rate of a disk-to-disk (dd) task should be revisited. Moreover, we present a simplified policy to compute incrementally the output generation rate within the reactive stages, and provide necessary changes to allow processing joins at multiple granularities. This section starts with an overview of the RPWJ algorithm, presents techniques to maintain metadata to support multi-granularity join processing, presents the optimal flushing policy, and analyzes the estimation of the output rate of the reactive tasks.

4.1 Overview of the RPWJ

In sliding window joins, as the window slides over time, tuples from a stream window should be invalidated periodically. This period depends on the sliding interval τ. In a continuously sliding window (i.e., τ is equal to the minimum unit or quanta of time), tuples from a window should be invalidated whenever a new tuple arrives in the opposite stream. Such an eager invalidation is infeasible while processing sliding window joins using a disk archive, as it would require a disk access for each incoming tuple. In such a scenario, lazy invalidation should be used. However, a tuple can be invalidated only if it is joined with all the tuples from the opposite window. Thus, the reactive stages (both md- and dd-stage) must be carried out immediately before an invalidation operation within a partition. Carrying out the reactive stages for all the partitions is a time consuming process. Thus the interval between two successive invalidations should be high. Note that even with a low stream arrival rate, such a lazy invalidation might result in a significant number of output tuples produced outside the time limit set by the window. [1] The RPWJ should be applied to window joins with a large τ. Having outlined the shortcoming of the RPWJ, we now describe the algorithm. RPWJ is carried out in three stages performed in an interleaved fashion (Figure 4.1).

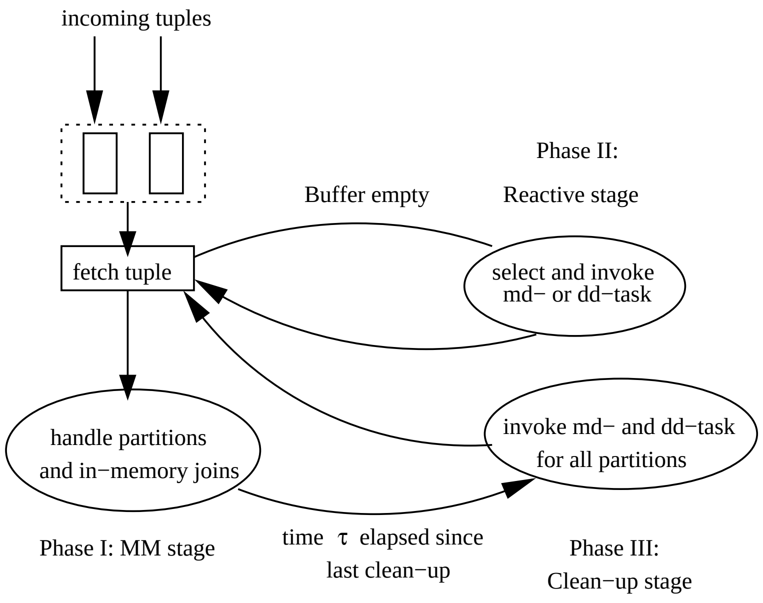

incoming tuples
Phase II:
Buffer empty
Reactive stage
fetch tuple
select and invoke md− or dd−task
handle partitions and in−memory joins
invoke md− and dd−task for all partitions
Phase I: MM stage
time τ elapsed since last clean−up
Phase III: Clean−up stage

Figure 4.1: Architecture of the RPWJ algorithm

The first stage joins the incoming tuples with the memory resident data of the opposite stream. RPWJ organizes the tuples within window W_i of stream S_i into partitions (Figure 4.2). Each partition j ($1 \leq j \leq n_{part}$, n_{part} being the total hash partitions) consists of two parts: a memory-resident part $W_i^{mem}[j]$, that stores the recent data within the partition, and a disk-resident part $W_i^{mem}[j]$, that stores data flushed to the disk due to memory limitations. When input tuples arrive from a source, the tuples are mapped to the respective partitions and joined with the memory-resident part of the partitions from the opposite window. If there is enough memory to store the tuples, then the tuples are stored on memory. Otherwise, one or more memory resident partitions are flushed onto the disk to make rooms for the incoming tuples. The first stage continues as long as there exist some tuples in the buffer to process; otherwise, the algorithm continues to the second phase.

The second phase chooses a reactive task from a pool of at most $3n_{part}$ possible tasks: there are at most $2n_{part}$ *md-tasks* ($W_i^{mem}[j] \bowtie W_{\bar{i}}^{disk}[j]$ for all $1 \leq j \leq n_{part}$ and $i = 1, 2$) and n_{part}

[1]Such tuples are reintegrated with the join results by a re-integration module outside the join module.

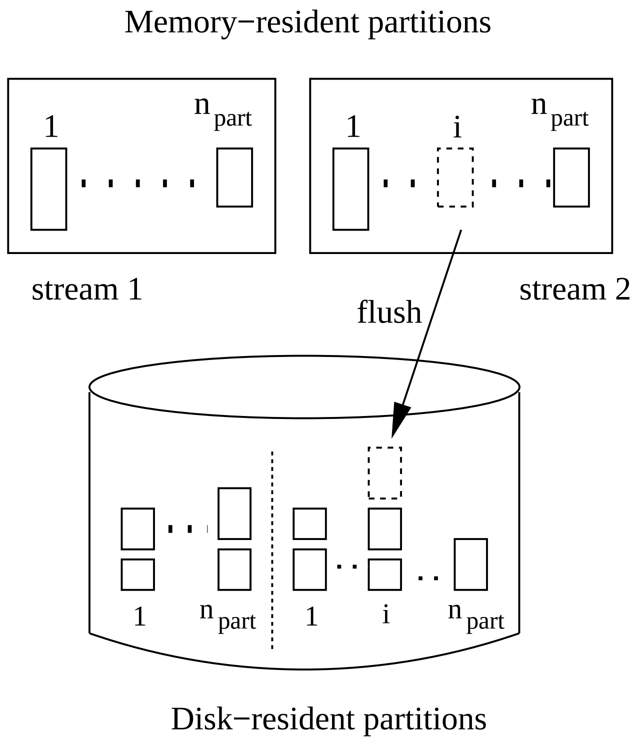


Figure 4.2: Memory stage of the join algorithm

dd-tasks ($W_1^{disk}[j] \bowtie W_2^{disk}[j]$ for all $1 \leq j \leq n_{part}$). The algorithm chooses a candidate task using a metric *Expected Output Rate*, which is defined as Er/Et, where Er is the expected number of output tuples generated by the task, and Et is the task execution time.

The third stage starts periodically at an interval of τ'. This stage invokes all $3n_{part}$ possible reactive tasks as stated above.

4.2 Processing Granularity

In all previous approaches, the incoming tuples are usually processed a tuple at a time. However, we observe that, when the buffer contains enough pending tuples, the join operation can be made more efficient by grouping the tuples in a block and processing the join operation at the granularity of the block. Such a block level join operation reduces the number of scans of the respective memory partition. In addition to reducing the overhead of partition scans, the block-level processing reduces the amount of meta-data to be kept within a block. We switch between block-level and tuple-level processing depending on the buffer occupancy: whenever the buffer occupancy is above H_{mark}, the algorithm processes at block-level; and while the buffer occupancy is below L_{mark}, it switches to tuple-level.

Each block contains a binary field *gLevel* which is set to 0 if the tuples within the block are processed at the tuple-level, and set to 1 otherwise. Duplicate elimination within output results, as described in the subsequent part of the section, requires each tuple or block to be associated with its arrival and departure (flush) timestamps. The departure timestamp (DTS) can be maintained at a block level irrespective if its processing granularity. However, in case of the tuple-level processing, the arrival timestamp(ATS) should be maintained at the tuple level. If tuples within a block b are processed at a granularity of a tuple, the block contains an array of timestamps. The i-th element of the array (ATS_i) contains the arrival timestamp of the i-th tuple within the block. In the subsequent part of the section, we consider only the block level processing; and the case for a block b with *gLevel* flag set to 0 can easily be extended with an extra level of iteration over each tuple t_i (i.e., i-th tuple within the block) within the block and checking against the relevant interval $[b.ATS_i, b.DTS]$.

4.3 Flushing Policy

The flushing policy, upon memory overflow, moves to the disk some memory resident tuples that are expected to produce the least number of output tuples until the next memory overflow. The victim partitions to be flushed are selected based on the arrival frequencies ($C_i[p]$) of the partitions. The arrival frequency ($C_i[p]$), which is maintained and decayed using a scheme similar to that in [27], indicates the volume of recent arrivals of tuples within partition p of stream i.

The flushing algorithm sorts the arrival frequencies of the partitions in descending order. The algorithm selects the arrival frequency $C_{i'}[p']$ with the minimum value, and flushes tuples from the p'-th partition of the opposite window onto the disk. Starting from the minimum arrival frequency, the algorithm continues to select and flush the successive partitions as long as the total volume of the tuples flushed onto the disk is less than N_{flush}. We flush the tuples at a partition level (i.e., don't flush a fraction of a partition) to render the task of the incremental maintenance of the Er-values simpler.

4.4 Duplicate Elimination in Reactive Stages

The reactive stage starts when there is no input tuple to process, and selects one of the *md- or dd-tasks* with the maximum output rate from a pool of $2n_{part}$ md-tasks and n_{part} dd-task.

The md-stage joins a memory partition $W_i^{mem}[p]$ with the disk partition of the opposite stream window ($W_{\bar{i}}^{disk}[p]$). Within this task, blocks from the disk partition are read and probed into the respective memory partition of the opposite window. The md-stage may generate redundant output tuples. There are only two ways in which an output tuple might be redundant: (1) the output tuple was generated in the mm-stage, or (2) the output tuple was generated in a previous md-task that involved $W_i^{mem}[p]$ and $W_{\bar{i}}^{disk}[p]$. Each memory partition $W_i^{mem}[p]$ maintains a list $T_{i,p}^{md}$ that stores the timestamps of the previous md-tasks involving the memory partition. As the window slides over time, the list is pruned to remove the tailing timestamps that fall outside the time window. So, after an invalidation, if the memory partition $W_i^{mem}[p]$ was involved in c md-tasks, the list $T_{i,p}^{md}$ contains c entries, the last entry $T_{i,p}^{md}[c]$ indicating the most recent md-task involving the memory partition. Thus output tuples resulting from a block $b_d \in W_{\bar{i}}^{disk}[p]$ and a block $b_m \in W_i^{mem}[p]$ are propagated (i.e., not duplicate) if none of the following two conditions are satisfied:

1. $[b_d.ATS, b_d.DTS]$ intersects $[b_m.ATS, b_m.DTS]$, which means that b_m and b_d found each other in memory, and are already joined.

2. $T_{i,p}^{md}[c] > b_m.ATS$ and $b_d.DTS < T_{i,p}^{md}[c]$, which means that b_m was joined with b_d is a previous md-stage.

It should be noted that the values $T_{i,p}^{md}[1]$, $T_{i,p}^{md}[2], \ldots, T_{i,p}^{md}[c-1]$ are not used in the duplicate elimination during a md-stage as stated above; however, these values are used during a dd-stage as stated below, and, therefore, should not be discarded.

The dd-task joins the two disk portions, from both the stream windows, of a partition p. The two disk partitions ($W_i^{disk}[p]$ and $W_{\bar{i}}^{disk}[p]$) are joined using the Blocked NLJ (Nested Loop Join) algorithm. The order of the tuples or blocks within a partition can't be altered. So, the Progressive Sort Merge Join (PSMJ) Algorithm, used while joining finite streams or relations, can no longer be applied in the scenario of a window join. We assume that the disk partition $W_i^{disk}[p]$ is used as the outer partition and the partition $W_{\bar{i}}^{disk}[p]$ as the inner one in the Block-NLJ algorithm. To reduce the number of disk IO operations during the join, we read the disk blocks as clusters and store these blocks in a reserved memory space. While joining a block $b_i \in W_i^{disk}[p]$ and a block $b_{\bar{i}} \in W_{\bar{i}}^{disk}[p]$, there might arise duplicate tuples in the following three possible ways. In the first case, the two blocks might have been joined in memory, during the mm-stage. This can be detected by simply checking, similar to a md-stage, the memory-alive intervals of the two blocks.

The second scenario is that the blocks were joined in a previous md-stage, which could be a join between $W_i^{mem}[p]$ and $W_{\bar{i}}^{disk}[p]$, or a join between $W_i^{disk}[p]$ and $W_{\bar{i}}^{mem}[p]$. Due to the symmetry, we consider the only former one. In this case, there must be a md-task when the block b_i was in memory (i.e., within the interval $[b_i.ATS, b_i.DTS]$), but $b_{\bar{i}}$ was flushed to the disk (i.e., the md-task happened after $b_{\bar{i}}.DTS$). If there are a total of c_i md-tasks since the last invalidation of the stream window, then their timestamps are given by $T_i^{md}[1], T_i^{md}[2], \ldots, T_i^{md}[c_i]$, as stated earlier in the section. If any of the timestamps lies within the interval $[b_i.ATS, b_i.DTS]$ and is greater that $b_{\bar{i}}.DTS$, then the blocks b_i and $b_{\bar{i}}$ were already joined in a previous md-stage.

As the third case, the two blocks b_i and $b_{\bar{i}}$ might have been produced during a preceding dd-task between $W_i^{disk}[p]$ and $W_{\bar{i}}^{disk}[p]$. To track whether two blocks are already joined in a dd-task, we keep a timestamp $lastDD_i^p$, which tracks the highest timestamp of the block scanned during the most recent dd-task, for each disk partition $W_i^{disk}[p]$ of stream i. If $b_i.DTS$ is less than $lastDD_i^p$ and $b_{\bar{i}}.DTS$ is also less than $lastDD_{\bar{i}}^p$, then these two blocks are already joined during the last dd-stage invoking the disk parts $W_i^{disk}[p]$ and $W_{\bar{i}}^{disk}[p]$.

4.5 Task Selection

As mentioned earlier in this section, the output rate of a md- or dd-task is given by Er/Et, where Er is the expected number of new results produced by a task, and Et is the task execution time. Predicting the output rate requires us to predict both Er and Et for all partitions of the stream windows. We denote the Er-value of an md-task between partitions $W_i^{mem}[p]$ and $W_{\bar{i}}^{disk}[p]$ as $Er_{i,p}^{mem}$, and that of a dd-task between $W_i^{disk}[p]$ and $W_{\bar{i}}^{disk}[p]$ as Er_p^{disk}. We start with the analysis of Et.

4.5.1 Analysis of Et

For a md-task joining $W_i^{mem}[p]$ and $W_{\bar{i}}^{disk}[p]$, the execution time Et is approximated by the time required to scan the disk partition $W_{\bar{i}}^{disk}[p]$—the time to access the in-memory partition $W_i^{mem}[p]$ is negligible compared to the access time of the disk partition. Blocks flushed to a disk partition, during optimal flushing, are stored sequentially in the disk; such a pool of blocks flushed together constitute a *run* as termed in [129, 42]. If blocks are flushed in the disk ($W_{\bar{i}}^{disk}[p]$) $x_{\bar{i}}$ number of times during the span of the stream window, it requires at least $x_{\bar{i}}$ random accesses. Suppose the size of the reserved memory space for disk I/O is M_{read}. Then, there will be at least $\frac{|W_{\bar{i}}^{disk}[p]|}{M_{read}}$ disk seeks and latencies (disk accesses) even if the blocks of the disk partition are stored sequentially (i.e., only one run). Thus there will be at most

$\left(x_{\bar{i}} + \frac{|W_{\bar{i}}^{disk}[p]|}{M_{read}}\right)$ disk accesses. Hence,

$$Et_{md} = \left(x_{\bar{i}} + \frac{|W_{\bar{i}}^{disk}[p]|}{M_{read}}\right).t_A + C_R * W_{\bar{i}}^{disk}[p] \tag{4.1}$$

Here, t_A is the disk access time, and C_R is the disk transfer rate expressed in sec/byte.

For a dd-task joining the partitions $W_i^{disk}[p]$ and $W_{\bar{i}}^{disk}[p]$, we need to fully scan the inner partition (say $W_{\bar{i}}^{disk}[p]$) for each iteration of the outer partition (i.e., $W_i^{disk}[p]$). As stated earlier, we share a reserved memory space of size M_{read} to read blocks from both the disk partitions. As analyzed in [63], for a block *nested loop join* involving two disk relations, the total disk I/O times is minimized when size of the blocks read from the disk at a time (i.e., *cluster size* [63]) is $M_{read}/2$ (i.e., the available memory is evenly divided between the relations). Hence, while scanning each of the disk partitions, we read $M_{read}/2$ blocks at a time. Thus with the cluster size $M_{read}/2$, the total time to scan a partition $W_i^{disk}[p]$ is given by

$$Et_{i,p} = \left(x_i + \frac{2|W_i^{disk}[p]|}{M_{read}}\right).t_A + C_R * |W_i^{disk}[p]| \tag{4.2}$$

Also, with the cluster size $\frac{M_{read}}{2}$, the inner partition $W_{\bar{i}}^{disk}[p]$ should be scanned $\frac{2|W_i^{disk}[p]|}{M_{read}}$ times. Therefore, Et for the dd-task can be written as,

$$Et_{dd} = Et_{i,p} + \frac{2|W_i^{disk}[p]|}{M_{read}} Et_{\bar{i},p} \tag{4.3}$$

Here, $Et_{i,p}$ and $Et_{\bar{i},p}$ are derived from equation 4.2.

4.5.2 Analysis of Er

As new tuples arrives, the expected number of output results from a reactive task changes over time. We provide a technique to estimate the expected join results in a md or dd-task. The Er-values for the partitions changes whenever a new tuple or block arrives in memory or a memory block is flushed onto the disk.

When a new tuple (in tuple-level processing) or block (in block-level processing) arrives in the partition p of stream i, the arriving tuple or block will be joined, during a subsequent md-task, with all the blocks in the disk portion of partition p of the opposite window $W_{\bar{i}}^{disk}[p]$. Thus, an arriving block b_m (considering the block-level processing) should generate $\sigma.|b_m|.|W_{\bar{i}}^{disk}[p]|$ output tuples, σ is the join selectivity within partition p. As stream tuples are partitioned using a hash function, we assume, like [129], that the join selectivity is same (σ) over all the partitions. Upon arrival of a block b_m, the expected output tuples ($Er_{i,p}^{md}$) for an md-task involving $W_i^{mem}[p]$ and $W_{\bar{i}}^{disk}[p]$ is increased by $\sigma.|b_m|.|W_{\bar{i}}^{disk}[p]|$. Whenever an md-task $W_i^{mem}[p] \bowtie W_{\bar{i}}^{disk}[p]$ is invoked, the value of $Er_{i,p}^{md}$ is reset to zero. Also, whenever a memory partition p of stream i ($W_i^{mem}[p]$) is flushed onto the disk, the respective Er-value (i.e.,$Er_{i,p}^{mem}$) is reset to zero. However, note that the output tuples to be generated in a md-task $W_i^{mem}[p] \bowtie W_{\bar{i}}^{disk}[p]$ will now be generated within a subsequent dd-task involving partition p. Thus, while flushing tuples from a partition, the value $Er_{i,p}^{md}$ is added to the Er_p^{dd}, which is the expected output tuples from a dd-task $W_i^{disk}[p] \bowtie W_{\bar{i}}^{disk}[p]$.

As disk flushing is carried out at the granularity of partitions, the task of maintaining the Er-values of the dd-tasks becomes simple. Such partition-level flushing does not affect the join performance, as the optimal performance is not restricted to a fixed value of N_{flush} but a wide range of values for N_{flush}. Note that the expected output rates of various tasks are compared among themselves while choosing an optimal reactive task, and their absolute values are not necessary. So, we can omit the join selectivity σ from the above calculation and use the values of $Er_{i,p}^{md}$ or Er_p^{dd}—which now provide the sizes of the Cartesian products—as an indication of the relative values of their output size.

4.6 Summary

This chapter presents an algorithm, called RPWJ, to process sliding window joins over data streams. The RPWJ algorithm is based on the RPJ that joins two finite data streams. We present numerous changes incorporated in the RPWJ algorithm. While joining two finite relations, the RPWJ algorithm, like the RPJ, maximizes the output rate by tuning the flushing policy and the reactive stages (i.e., md- or dd-task) based on the data distribution and tuple arrival patterns of the relations. We use the RPWJ algorithm as a baseline one to compare with the performance of our proposed algorithm in the subsequent chapter.

Chapter 5

Adaptive, Hash-partitioned Stream Joins

As observed in the EWJ algorithm proposed in Chapter 3, the CPU becomes the bottleneck from a high arrival rate. In this chapter, we propose a partition-based approach to process sliding window joins that minimize the processing overhead. The Adaptive, Hash-partitioned Exact Window Join (AH-EWJ) algorithm proposed in this chapter extends the hashjoin to spill tuples in the partitions onto the disk and adjust the partition sizes depending of productivity and stream arrival rates.

5.1 Introduction

In a DSMS processing a large number of join operators, both the CPU and the memory becomes the limited resources. Numerous techniques based on load shedding have been developed identifying the resource limitations. In CPU limited environments, tuples are added to the streams windows; but the added tuples may be either omitted joining with the stream windows [6] or be joined with a selected fraction of the stream windows [52]. In both CPU and memory limited environment, random [130] or semantic load shedding [40, 125] drops tuples from the stream buffers before reaching at the stream buffers. In this chapter, we propose the AH-EWJ algorithm that takes a different approach. We address the problem of providing accurate results to sliding window join queries over streams with time varying or bursty arrivals of the tuples. During bursts of arrivals, both the CPU or the memory might be overloaded, and instead of dropping tuples, the proposed algorithm spills data onto the disk. Contrary to load shedding, the goal here is to smooth the loads over time using a disk storage.

Using disk storage as an archive, this chapter present the AH-EWJ algorithm that endeavors to smooth the load by spilling a portion of both the window blocks onto the disk. We propose a framework for processing disk-based sliding window joins. The proposed algorithm merges the disk resident data periodically with the in-memory blocks during a phase called *disk probing*. This approach merges the flushed tuples onto one disk partition and amortizes a disk-access over a large number of input tuples, thus eliminating small, random disk I/O; it improves memory utilization by employing *passive removal* of the blocks from the stream window. Also, this proposed algorithm dynamically adjusts the stream windows and fine tunes the hash-buckets within each stream (during *passive removal*) based on the stream arrival patterns. To increase the output generation rate, the AH-EWJ algorithm employs a generalized framework to manage the memory blocks forgoing any assumption about the models (unlike previous works e.g. [125]) of stream arrival. In summary, the key contributions of the chapter are as follows:

1. We propose a disk-based join algorithm AH-EWJ that, given a limited memory, processes sliding window joins over arbitrary sequences of streaming data. The proposed algorithm produces correct output by spilling data onto the disk and retrieving/joining the spilled data within the time bound set by the window.

2. We propose a disk probing policy that, on one hand, maximizes disk efficiency by amortizing a disk scan over a number of tuples. On the other hand, the same policy maintains the productive blocks within memory, thus maximizing output rate. The proposed technique can cope with arbitrary patterns of stream arrivals.

3. We propose a methodology to adjust both the window and the partition sizes based on the arrival rates of the streams. Our proposed scheme avoids disk dump whenever the windows can be incorporated within the memory.

4. We present detailed experimental studies showing the effectiveness of our techniques.

The rest of the chapter is organized as follows. Section 5.2 defines the problem and provides an overview of the proposed algorithm. Section 5.3 describes the techniques to manage memory blocks, the algorithm to integrate periodically the disk and memory blocks (disk probing), and finally the AH-EWJ algorithm. Section 5.4 describes experimental methodologies in details, and presents the experimental results comparing the performance of the proposed algorithm (AH-EWJ) with that of the baseline one (RPWJ). Section 5.5 concludes the chapter.

5.2 Problem Definition and Solution Overview

The basic join operator considered in this chapter is a sliding window equi-join between two streams S_1 and S_2 over a common attribute A, denoted as $S_1[W_1] \bowtie S_2[W_2]$. The output of the join consists of all pairs of tuples $s_1 \in S_1, s_2 \in S_2$ such that $s_1.A = s_2.A$, and $s_1 \in S_1[W_1]$ at time $s_2.t$ (i.e., $s_1.t \in [s_2.t - W_1, s_2.t]$) or $s_2 \in S_2[W_2]$ at time $s_1.t$ (i.e., $s_2.t \in [s_1.t - W_2, s_1.t]$. We assume that the amount of memory available for storing the join operator states or windows is less than the size of the windows. Hence, a portion of each of the stream window W_i resides on the disk. The proposed algorithm (AH-EWJ) is based on the hashing methodology. Tuples in the stream windows are mapped to one of the n_{part} partitions, using a hash function $\mathcal{H}$ that generates an integer in the range of $[1, n_{part}]$. Hence, the join algorithm should keep a portion of each of the hash partitions (or , buckets) into the memory and spill the remaining portion onto the disk. The join operator has buffers attached with its input streams. The join operator fetches tuples from the input buffers, processes the join, and sends the output as a stream; it must ensure that each result tuple is generated exactly once.

The straightforward approach to the sliding window join over unbounded streams is to use the existing algorithm XJoin [133] (or, its variants, i.e., RPJ [129]), and invoke the clean up or reactive stages (i.e., disk-to-memory and disk-to-disk) periodically. But, this would result in significant processing and disk overhead as the scan of a disk or memory segment might occur for a few tuples within a partition. In this approach, the number disk I/Os for both the invalidation (of the stream windows) and the disk dump operation would also be $O(n_{part})$. The performance of this algorithm degenerates drastically in situations with low sliding interval, as both the clean-up of the disk resident data and the tuple invalidation should occur frequently, rendering the above approach impractical in real time processing of sliding window joins over data streams.

In addition to the above limitations, our approach is motivated by the following observations on the performance of XJoin and its variants: First, these approaches attains low utilization of memory. When, the memory is full, XJoin flushes tuples (N_{flush}) from a few selected partitions. Hence, on the average, a portion of memory equivalent to the size of $\frac{N_{flush}}{2}$ tuples remains unutilized. This unutilized memory can be decreased by selecting a low N_{flush}, but this in turn increases the I/O overhead as it does not amortize a disk write over a large number of input tuples. Hence, the proposed approach should eliminate such holes in the memory, and make the disk flush efficient by eliminating frequent and small disk writes.

Secondly, duplicate elimination is a complex process with significant processing and storage overhead. The reactive stage, that retrieves disk resident blocks, is invoked frequently to increase the join output rate. For a partition from each stream, a list of timestamps is maintained which records the times the memory-to-disk task (one of the two tasks in a reactive stage) is invoked. Each tuple in the disk is associated with a time interval recording the times the tuple arrived into the memory and flushed onto the disk. To eliminate duplicate output tuples, the above list from one or two partitions (depending on the task in the reactive stage, i.e., disk-to-memory or disk-to-disk) of two streams should be scanned for each result tuples. Hence, a simplified approach to duplicate elimination is necessary that at the same time harnesses the advantage (i.e., increased output generation rate) of the reactive stage in XJoin or its variants.

We propose an algorithm called *Adaptive, Hash Partitioned Exact Window Join* (AH-EWJ) that incorporates the above observations. The algorithm computes sliding window joins between two streams without compromising the output accuracy. Existing algorithms shed the load by dropping tuples selectively. Contrary to these algorithms, we attempt to defer the load during high workload, and process the deferred load during the period of low workload. Thus, AH-EWJ aims at smoothing the load using the disk as secondary storage. The proposed algorithm AH-EWJ is based on the framework given in Figure 5.1 showing the organization of tuples within a stream window W_i. For each stream $S_i (i = 1, 2)$, AH-EWJ stores the relevant tuples (window states) in the memory and on the disk. We denote the portion of the window $S_i[W_i]$ residing in memory and disk as W_i^{mem} and W_i^{disk} respectively. We assume that the sizes of a memory page and a disk block are equal, and use the terms interchangeably. We arrange the tuples into pages (or blocks) and process the tuples at the granularity of a page or block. In addition to the timestamp of the tuples within a page, each page or block contains a unique sequence number (or timestamp). All tuples within a block are homogeneous in the sense that they are probed against the same number of blocks from the window of the opposite stream $(S_{\bar{i}}[W_{\bar{i}}])$.

AH-EWJ eliminates a large number of small, random I/Os (one for each partition) by merging the flushed blocks into one disk segment and performing disk read on this unified segment. The memory segment of a window W_i is divided into an *invalidation segment* and a *generative segment*. An invalidation segment is allocated to reduce disk I/O while expiring the tuples from the stream window which makes the tuple expiration efficient; the generative segment stores the more productive blocks to maximize the output throughput. Generative segment has two parts: *frequent segment* and *recent segment*. An incoming tuple s in stream S_i is joined with the memory resident portion of the hash partition from the opposite window (i.e., $W_{\bar{i}}^{mem}[\mathcal{H}(s)]$). When the stream windows can not be accommodated within the join memory, a portion of a stream window (i.e., N_{flush}^{ah-ewj} blocks) is flushed or dumped onto the disk. A block

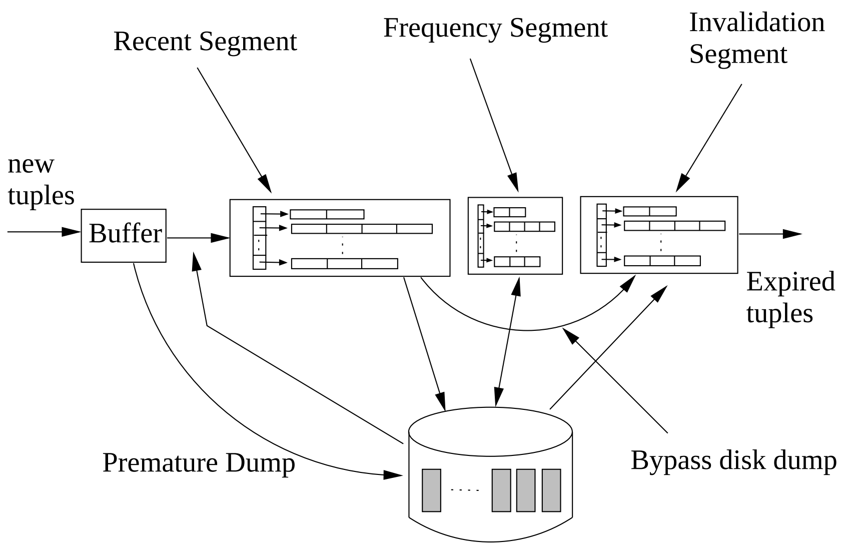


Figure 5.1: Organization of incoming tuples within a window

(from the recent segment of W_i) is dumped onto the disk only when it is joined with all the blocks within the disk portion of the window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{disk}$). Such a dump activity requires periodic scanning of the disk to retrieve the blocks from $W_{\bar{i}}^{disk}$ and joining with the pages to be dumped—an action referred to as *disk probing*. The dump operation merges a number of blocks (i.e., N_{flush}^{ah-ewj}) from all the hash partitions and stores in the respective disk segment invoking only one disk write. AH-EWJ keeps the flushed blocks in the recent segment and removes the blocks only on demand. This eliminates the holes in the memory and achieves high memory utilization. On the other hand, frequent segment stores the blocks with long term frequency of output generation. We explain the rationale and management of the segments in the subsequent sections. In order to cope with the unpredictable variations in arrival patterns, blocks within the frequent segment should be updated in a regular fashion. This update operation is carried out during disk probing. Moreover, based on the stream arrival rates, AH-EWJ adjusts memory allocated to the stream windows. AH-EWJ also fine tunes the sizes of the partitions (or buckets) based on the arrival patterns. We describe the details of the algorithm in the subsequent section.

5.3 Exact Join

In this section, we describe in details the join algorithm for processing sliding-window join for streaming data. The algorithm consists of four major sub-tasks: invalidation of the tuples, maintaining blocks in generative segments, adapting the memory allocation both within and

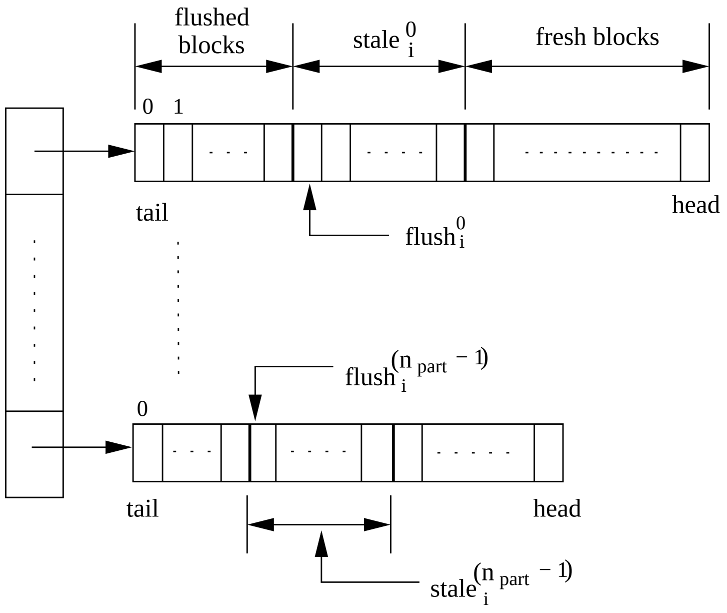


Figure 5.2: Data structure of a Recent Segment of a Stream Window

across the stream windows, and probing the disk periodically to join the disk resident data with the incoming data. Before presenting the algorithm we illustrate the key ideas encompassing the major subtasks. The generative segment of a window W_i is divided into two segments: Recent segment (W_i^{rec}) and Frequent segment (W_i^{freq}). So, the memory portion of each window W_i has three segments in total: invalidation segment (W_i^{inv}), Recent segment and Frequent segment. Each of the segments is maintained as a hashtable with n_{part} partitions or buckets. We denote the j-th partition of a segment W_i^{seg} as $W_i^{seg}[j]$, where $seg \in \{rec, freq, inv, mem\}$.

5.3.1 Memory Stage

The incoming tuples in stream S_i are mapped to the respective partitions using a hash function $\mathcal{H}$ and added to the head block of the partition in the recent segment $W_i^{rec}[p][H]$. Each newly added block within a stream is given a unique number denoted as *block number*. Other than the block number, every block is associated with a *productivity value* that records the number of output tuples generated by the tuples in that block. The block number is used in eliminating redundant output tuples as described in the subsequent part of this section. The arriving tuples are joined with the respective partition in the memory segment of the opposite window $W_{\bar{i}}^{mem}[p]$. Blocks (in stream S_i) are added to the recent segment at one end (i.e., head), while the blocks from the other end (i.e., tail) are stored to the disk during a disk dump. In our join algorithm, we maintain the following constraint:

Constraint **a**: *At a particular time, the blocks from recent segment of stream S_i should join with all the blocks in the disk portion of window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{disk}$) before being dumped to the disk (during mature dump)*

Blocks in the recent segment are categorized into three types: flushed blocks, stale blocks and fresh blocks. The fresh blocks are newly arrived blocks that have not yet participated in the disk probing. The stale blocks have already participated in the disk probing, but have not been flushed onto the disk. The flushed blocks have already been dumped onto the disk, and, therefore, can be deallocated to accommodate newly arrived blocks. During the disk probing phase as described in subsection 5.3.2, blocks retrieved from the disk segment (of the opposite stream) are joined with the *fresh blocks* in the recent segment. After the disk probing, the fresh blocks are marked as stale ones. Blocks from the recent segment are evicted upon the arrival of new tuples, and the victim blocks are selected adaptively based on both the stream arrival patterns and productivity values (described in subsection 5.3.5). A victim partition within a recent segment usually evicts a flushed block. However, as described in the subsection 5.3.5, free blocks from the invalidation segment can sometimes be evicted. Figure 5.2 shows the structure of a stream window. Here, $flush_i^p$ is the location of the first stale block in a partition p of recent segment of stream S_i, and $stale_i^p$ is the number of stale blocks in the same partition. Blocks in the locations preceding $flush_i^p$ are the flushed blocks; those in the range $[flush^p, flush_i^p + stale_i^p - 1]$ are the stale blocks; and those in the range $[flush_i^p + stale_i^p, |W_i^{rec}[p] - 1|]$ are the flesh blocks. After disk probing, the value of $stale_i^p$ increases; whereas, after a disk dump, the value of $stale_i^p$ decreases and that of $flush_i^p$ increases, setting the periphery of the flushed blocks.

It should be clear that a disk dump in stream S_i merges the N_{flush}^{ah-ewj} number of stale blocks from all the partitions in the recent segment W_i^{rec} and flushes the tuples onto the disk. During a disk dump, blocks from a partition are stored sequentially, that minimizes partition switches while joining the blocks with the respective partition during the disk probe (we describe the detailed technique in Subsection 5.3.4). Note that, the blocks dumped onto the disk (i.e., flushed blocks) are kept in the partitions, and these blocks are joined with the blocks/tuples arriving in the opposite stream. The flushed blocks are removed only on demand. Such a lazy removal of the flushed blocks achieves high memory utilization increasing the output generation rate. Without the lazy removal, the recent segment might fluctuate between overflow and empty states: on the average, half of the recent segment would remain unutilized.

5.3.2 Disk Probing

In this subsection, we present the basic techniques and data structures involved with disk probing.

Frequent Segment Update

The decision about placement or replacement of a block in Frequent segment is based on its productivity value. This decision is made periodically after a certain interval. The productivity values of the blocks are decayed during this update stage. The feasible time to carry out this update step is during the disk probing or mature dump when the all the disk blocks are scanned. During the update stage, as the disk blocks are scanned, a block is brought into the Frequent segment if its productivity exceeds that of a block already in Frequent segment at least by a fraction (ρ). The block having the minimum productivity value among the blocks in the Frequent segment is replaced.

Productivity Table

All the blocks within a window need to store the productivity values. Updating this value for a memory resident block is easy; however, updating the productivity value for a disk resident block requires one disk write to update the corresponding block. To make this update of productivity values efficient we maintain the productivity values of all the disk resident blocks in memory as a list of tuples <*block number, productivity*>, and use the term productivity table to refer to this list. Such a table consumes little space and might be stored in the memory. The productivity values of the blocks in a Disk segment are updated during disk probing.

Eliminating Redundant Tuples

A block b_i^p evicted from the Frequent segment of window W_i is already joined with the fresh blocks in the Recent segment of the stream $W_{\bar{i}}$. If the evicted block is not already scanned on the disk, it will be read and be joined with the fresh blocks in the Recent segment of window $W_{\bar{i}}$. To prevent this duplicate processing, we maintain a list E_i containing the block numbers of evicted blocks. If an incoming block from the disk is contained in E_i, we omit processing that block. List E_i is reset to *null* at the end of the update stage. The productivity values of all the blocks in the disk segment and the frequent segment are decayed (using a factor α $(0 < \alpha < 1)$) after the update stage.

As described earlier, blocks from the Recent segment are removed on demand. Such passive removal of the blocks might lead to duplicate output generation. Let us consider a block b_i^j in partition $\mathcal{H}(b_j^i)$ of the recent segment W_i^{rec}. The block b_i^j joins with a block $b_{\bar{i}}^k \in W_{\bar{i}}^{rec}[\mathcal{H}(b_j^i)]$. Later b_i^j participates in mature dump and is stored on the disk. However, due to the passive removal of the blocks from the Recent segment, the same block (b_i^j) remains in memory as a stale block. Now, when block $b_{\bar{i}}^k$ of the opposite window participates in mature dump, it finds on the disk the block b_i^j already joined in previous step. We use the sequence number of a block, denoted as *block number*, in solving this issue: every incoming block is assigned an unique number from an increasing sequence. Every block b_i^k in partition $\mathcal{H}(b_i^k)$ of the Recent segment W_i^{rec} stores the minimum block number (*minBN*) among the blocks, from the same partition of the Recent segment of the opposite window, that b_i^k joins with. When b_i^k participates in mature dump, it joins with a block $b_{\bar{i}}^p \in W_{\bar{i}}^{rec}[\mathcal{H}(b_k^i)]$ if the block number of the block $b_{\bar{i}}^p$ is less than the *minBN* value of the block b_i^k, i.e., $b_{\bar{i}}^p.BN < b_i^k.minBN$. As b_i^k is already in W_i^{mem}, any block in $W_{\bar{i}}$ arriving after $b_i^k.minBN$ is already joined with b_i^k. So, during disk probing any block $b_{\bar{i}}^p \in W_i^{disk}$ having $b_{\bar{i}}^p.BN \geq b_i^k.minBN$ can be omitted.

On the other hand, we observe that any block b_i^p in a Frequent segment of window W_i (i.e.,W_i^{freq}) is already joined with the corresponding partition in the memory segment of window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{rec}$, $W_{\bar{i}}^{inv}$). When a block is brought into the Frequent segment, it is already joined with the blocks in disk segment of window $W_{\bar{i}}$, i.e, $W_{\bar{i}}^{disk}$ (refer to **Constraint a** stated earlier), and that block is joined with $W_{\bar{i}}^{rec}$ immediately before being placed on the Frequent segment. Now, any incoming block in window $W_{\bar{i}}$ is joined with b_i^p. Moreover, if any block is dumped on the disk, it has already found b_i^p on memory and has been joined with that block. No issue of duplicate processing other than the one mentioned above arises with frequent segment. Hence, blocks in frequent segment need not store any extra information.

Probing Algorithm

Function DISKPROBE(), as shown in Algorithm 3, is invoked periodically when the number of fresh blocks within a Recent segment exceeds a predefined threshold N_{rec}^{min} (cf. 5.3.5). The function scans the disk retrieving blocks in $W_{\bar{i}}^{disk}$ and joins the blocks with those in W_i^{rec}. Blocks in the Frequent segment of window $W_{\bar{i}}$ (i.e., $W_{\bar{i}}^{freq}$) are updated periodically, and this update phase is carried out with disk probing. Moreover, this *update interval* might be longer than the *probing interval*—the interval between two consecutive disk probes in a stream. So, the parameter $uFlag$ is used to indicate when to update the relevant Frequent segment (i.e.,$W_{\bar{i}}^{freq}$). To eliminate duplicate tuples in output, an incoming block from the disk is checked against the condition in line 7 before joining with the fresh blocks in W_i^{rec}. The

Algorithm 3 DISKPROBE(W_i^{rec}, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, $uFlag$)

Input: W_i^{rec}, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, $uFlag$
Output: results of stream joins; update frequent segment if $uFlag$ is True

```
 1: position disk head to the first block in $W_{\bar{i}}^{disk}$
 2: $E_{\bar{i}} \leftarrow null$
 3: while exists new block in $W_{\bar{i}}^{disk}$ do
 4:     read block $b_{\bar{i}}^m$ from disk
 5:     $p \leftarrow \mathcal{H}(b_{\bar{i}}^m)$
 6:     for $n \leftarrow (flush_i^p + stale_i^p)$ to $|W_i^{rec}[p]|-1$ do
 7:         if $b_{\bar{i}}^m.BN < b_i^n.minBN$ and; $b_{\bar{i}}^m \notin W_{\bar{i}}^{freq}[p]$; or $b_{\bar{i}}^m \notin E_{\bar{i}}$ then
 8:             join $b_i^n$ with $b_{\bar{i}}^m$
 9:             update $b_{\bar{i}}^m.prod$ and $b_i^n.prod$
10:             $b_i^n.minBN \leftarrow null$
11:             if $uFlag$=TRUE then
12:                 $bn \leftarrow$ UPDATEFREQSEGMENT($W_{\bar{i}}^{freq}, b_{\bar{i}}^m$)
13:                 if $bn \geq 0$ then
14:                     insert $bn$ into $E_i$
15:                 end if
16:             end if
17:         end if
18:     end for
19: end while
```

Algorithm 4 UPDATEFREQSEGMENT(W_j^{freq}, b_j^m)

Input: a frequent segment W_j^{freq}, and a block b_j^m)
Return: the evicted block number
Local variables: bn, p

```
 1: $b_{min} \leftarrow 0$ MINPRODBLOCK($W_j^{freq}$)
 2: $bn \leftarrow -1$ {a negative number}
 3: $p \leftarrow \mathcal{H}(b_j^m)$
 4: if $b_j^m.prod > (1+\rho)b_{min}$ then
 5:     $bn \leftarrow b_{min}.BN$
 6:     Evict $b_{min}$ from $W_i^{freq}[\mathcal{H}(b_{min})]$
 7:     copy $b_i^m$ to $b_{min}$
 8:     insert block $b_i^m$ into $W_i^{freq}[p]$
 9: end if
10: return $bn$
```

first part of the condition, as described earlier in this section, ensures that the disk block was not joined with the memory resident block (from W_i^{rec}) before being dumped onto the disk, while the second part checks if the incoming block is already in the Frequent segment or is recently evicted from the Frequent segment. In line 9, the productivity values of the blocks are updated on the productivity table avoiding expensive disk writes. Line 12 invokes the function UPDATEFREQSEGMENT, which updates the Frequent segment $W_{\bar{i}}^{freq}$. If the input block is stored within the frequent segment, the function returns the non-negative block number of the evicted block.

5.3.3 Invalidation

As the window slides over time, tuples from the tail of the window should be discarded continuously. To reduce the disk access time in eliminating tuples, we reserve a few blocks in memory to store the tail of the window and term the blocks as an *invalidation segment*. Without this invalidation segment (W_i^{inv}), in the worst case, we need to access the disk for each incoming tuple. Using this invalidation segment, we discard the tuples from the memory blocks whenever the window slides over time; the disk segment is accessed only when the invalidation segment of the corresponding window can accommodate a chunk of blocks B^{win} from the disk segment termed as a Basic Window.

5.3.4 Disk Dump

As illustrated earlier, during disk dump, blocks from all the partitions of a recent segment are merged and stored in the disk as a unified disk segment of the corresponding window. Now, if the blocks in the disk segment are not sequenced according to their time-stamps, the process of the invalidation/expiration of tuples might be affected. To realize the scenario, let us consider a disk segment where blocks with tuples of higher timestamps (i.e., recent ones) are followed by those with lower timestamps (i.e. older ones). Hence, the invalidation segment gets filled with the tuples having higher time-stamps (i.e., more recent ones). Now, the older tuples can not be brought into memory, and thus invalidated, unless all the younger tuples within the invalidation segment are expired. Such a blocking due to disorder of the blocks within the disk segment unnecessarily halts the expiration of the older tuples, that also delays their processing or tuple generation time. To alleviate the problem, two methods can be used: (1) stale blocks from the recent segment are sequenced according to their block number, (2) blocks from the partitions can be sequenced by fetching one block at a time, from the partitions, in a round

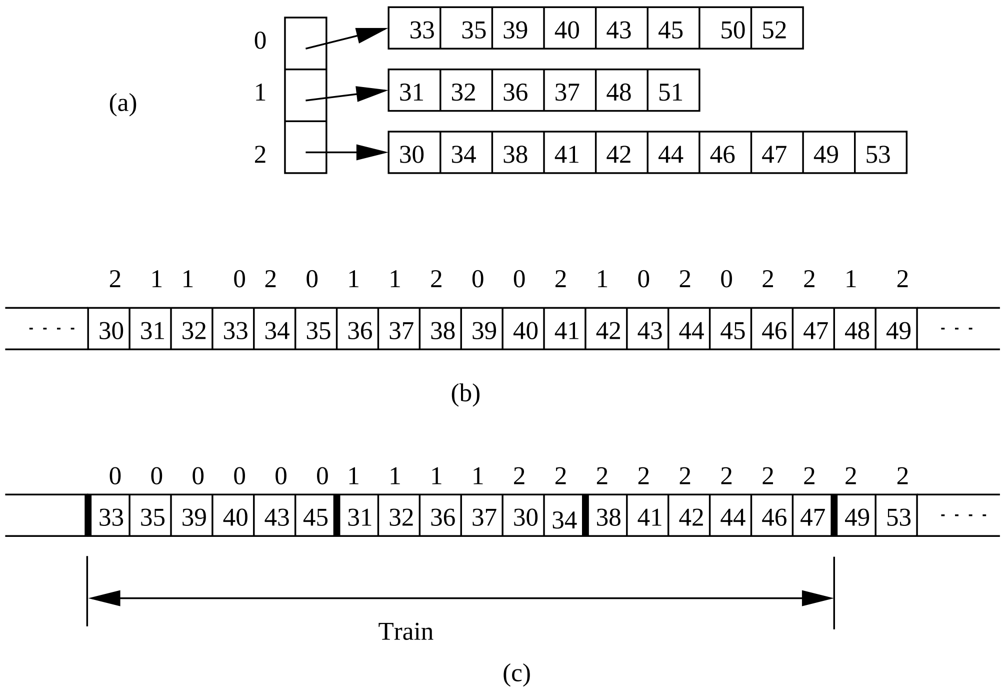


Figure 5.3: Mechanism to dump the stale blocks from a recent segment. (a) shows the stale blocks in the hash table. (b) sequence of blocks dumped according to their block number (c) dumping with bounded disorder using the notion of a *train*

robin fashion. However, both of the approaches result in cache thrashing while joining the blocks in the disk segment with the opposite window. This is due to the changes in partitions for each subsequent blocks.

Based on the above observations, we propose a technique to order the blocks, that stores the blocks from the same partitions in a sequence. Storing the blocks from the same partitions in a succession reduces the cache thrashing (at L1 and/or L2 caches) due to the frequent switching of the partitions while joining the blocks from the disk segment. Figure 5.3 illustrates the technique to dump the blocks. The blocks are identified by the *block number* BN. All the stale blocks with number less than 30 are either dumped onto the disk to transferred to the invalidation segment (due to an empty disk segment). Now, arranging the blocks according to the block numbers results in large number of partition switches (i.e., successive blocks belong to different partitions) as shown in Figure 5.3(b). Here, the numbers above the list indicates the partition numbers of the blocks. On the other hand, our approach based on bounded disorder causes only a few partition switches. Here, blocks from the same partition are stored in a sequence. However, storing a large number of tuples from the same partition might lead to disorder, which in turn might block the invalidation segment. We use the notion of a *train size*

Algorithm 5 DISKDUMP(int i, int $N_f^{ah-ewj}lush$)

Input: i=stream index; $N_f^{ah-ewj}lush$=total tuples to flush
Effect: dumps blocks onto the disk
Global variables: $W_i; i = 0, 1; trainSize, trainLimit_i, dumpC_i$

```
 1: if $dumpC_i \geq trainLimit_i$ then
 2:     $trainLimit_i \leftarrow trainLimit_i + trainSize$
 3: end if
 4: for $i \leftarrow 1$ to $n$ do
 5:     if BN of the block at location $flush_i^p$ is equal to $trainLimit_i$ then
 6:         $p \leftarrow$ next partition with BN of the block at location $flush_i^p$ less than $trainLimit_i$
 7:     end if
 8:     flush the block at location $flush_i^p$ to disk
 9:     increment $flush_i^p$ and decrement $stale_i^p$
10:     increment $dumpC_i$
11: end for
```

to bound the disorder within the block numbers. The number of blocks that might be dumped in a sequence is determined by the *train size*. At the start of every *train*, the upper limit of the block number of the train for stream i (i.e., $trainLimit_i$) is determined. Blocks from a partition are dumped until a block with BN higher that upper limit is encountered; in that case, the blocks from the next partition is dumped. Figure 5.3(c) shows the blocks as dumped onto the disk. Here, the train size is 18, and the flush size N_{flush}^{ah-ewj} is 6. As the lowest BN among the stale blocks is 30 (i.e., 30 might be the upper limit for the previous *train*), the upper limit for the current *train* is 48. Thus each *disk dump* within the *train* arranges blocks from the same partition until no more blocks in the partition is left or the next block in the partition has a BN not less than 48. Note that, during 2nd dump, blocks from the partition 1 are flushed until block 48 is encountered, at which point blocks from the partition 2 are flushed. The boundaries of the blocks within a disk dump are indicated by the bold lines. The partitions are switched in a cyclic fashion once the upper limit is reached. At the end of the 3rd dump, a new train starts with an upper limit 66 (=48+18). Algorithm 5 shows the pseudo-code for disk dump operation.

5.3.5 Adapting Window and Bucket Sizes

In the join scheme, sizes of the invalidation and the frequent segments are kept constant; the remnant join memory (M_{free}) is allocated between the recent segments, that change their sizes depending on stream arrival rates. Upon arrival of a new block, if no free blocks are available within the recent segment, the recent segment either evicts a block from itself or receives a block from the opposite window. To ascertain the memory allocation (between the recent segments) so as to maximize the join output rate, we use an equation similar to one in [79]. Let us start with a formula that captures the overall output rate of the join operation. Given the arrival rates of the streams λ_1 and λ_2, the smaller value of the selectivity factors of the recent segments σ, the output rate r_0 can be approximated as

$$r_0 = \sigma(\lambda_1|W_2^{rec}| + \lambda_2|W_1^{rec}|)$$
$$\text{Here, } |W_1^{rec}| + |W_2^{rec}| = M_{free}$$

The above equation can be rewritten as,

$$\begin{aligned} r_0 &= \sigma(\lambda_1(M_{free} - |W_2^{rec}|) + \lambda_2|W_1^{rec}|) \\ &= \sigma(\lambda_1 M_{free} + |W_2^{rec}|(\lambda_2 - \lambda_1)) \end{aligned}$$

Without loss of generality, let use assume that $\lambda_2 > \lambda_1$; hence, from the above equation, it is easy to deduce that the best strategy is to allocate most memory to the window corresponding to the lower input rate. To amortize the disk access cost over a large number of fresh tuples, while invoking disk probe, we set the minimum size of a recent segment as $Mr_{rec}(0 < r_{rec} < 0.5)$, equivalent to N_{rec}^{min} blocks; Here, M is the total join memory. Here, N_{rec}^{min} is greater than N_{flush}^{ah-ewj}.

To capture the instantaneous arrival rates of a stream, we use a metric termed as arrival frequency ($C_i; i = 1, 2$) that is decayed at the elapse of an interval or epoch (t_{epoch}. The decaying scheme [27] works as follows: within an epoch, the arrival frequency of a stream is increased upon the processing of a block from the stream; whereas, at the end of every epoch, the metric C_i is updated as αC_i. Stream i yields a block to the recent segment of stream $\bar{i}$, if the following conditions hold,

$$\begin{gathered} C_i \geq (1+\rho)C_{\bar{i}} \\ |W_{\bar{i}}^{rec}| > B_{rec}^{min} \wedge \sum_{p=0}^{n_{part}-1}(flush_i^p + stale_i^p) \geq 0 \end{gathered} \tag{5.1}$$

In the above inequalities, ρ is a user defined parameter that controls the aggressiveness of eviction of a block within the recent segment. Within the recent segment, a victim partition is selected based on the productivity metric ($prod_i^x$) of the partitions: a partition with the lowest productivity metric is selected that yields a tailing, flushed block.

Algorithm 6 presents the function for evicting a block from a recent segment. The join

Algorithm 6 EVICTRECENTBLOCK(int k)

```
Input: k = stream index
Return: a freed block $b$
Global variables: $W_i; i = 0, 1$
1:  $i \leftarrow k$
2:  if CANSUPPLY( $\bar{k}$ ) then
3:      $i \leftarrow \bar{k}$
4:  end if
5:  if $flush_i = 0$ then
6:      if $W_i^{disk} = \phi$ and $free(W_i^{inv}) > 0$ then {Bypass the Disk segment}
7:          $p \leftarrow$ partition having the block with the lowest $BN$
8:          remove tailing block $b$ in $W_i^{rec}[p]$
9:          copy $b$ to $W_i^{inv}[p]$
10:         decrement $stale_i^p$
11:     else
12:         $n \leftarrow min(B_{rec}^{min}, \sum_{p=0}^{n_{part}-1} stale_i^p)$
13:         DISKDUMP(i,n)
14:     end if
15: end if
16: if $flush_i \neq 0$ then
17:     $p \leftarrow \underset{x \mid stale_i^x > 0}{\arg\min} (prod_i^x)$
18:     remove tailing block $b$ in $W_i^{rec}[p]$
19:     decrement $flush_i^p$
20:     UPDATEFREQSEGMENT($W_i^{freq}$, b)
21: end if
22: return $b$
```

Notations	Description
S_i, $S_{\bar{i}}$	stream i and opposite to i, respectively
W_i, $W_{\bar{i}}$	Window of stream S_i and $S_{\bar{i}}$, respectively
W_i^{disk}	Disk portion of window W_i
W_i^{rec}	recent segment of window W_i
W_i^{freq}	Frequent segment of window W_i
W_i^{inv}	invalidation segment of window W_i
W_i^{mem}	memory portion of window W_i ($W_i^{rec} + W_i^{freq} + W_i^{inv}$)
$W_i^{seg}[p]$	partition p of $seg \in \{rec, freq, inv\}$
$W_i^{rec}[p][H]$	block at the head of $W_i^{rec}[p]$
b_i^p	block p in window W_i
$b_i^p.minBN$	minimum block number from $W_{\bar{i}}^{disk}$ & $W_{\bar{i}}^{rec}$ that $b_i^p \in W_i^{rec}$ joins with
$b_i^p.prod$	Productivity of block b_i^p
ρ	parameter to evict a block within a frequent segment and a recent segment

Table 5.1: Notations and the system parameters (AH-EWJ)

algorithm invokes the function whenever a recent segment is full and new block is desired to store the incoming window-tuples. The function selects, based on the criteria presented in equation 5.1, a stream that should supply a block (line 1–4). The selected stream usually supplies an available flushed block; if the selected stream has no flushed blocks, the function either flushes the stale blocks onto the disk (creating a pool of flushed blocks) or copy a stale block to the invalidation segment. The former scenario arises if the respective disk segment is not empty or the invalidation segment has no free blocks; therefore, a pool of stale blocks are dumped onto the disk converting them to the flushed ones (line 12–13); whereas, in the latter situation, if the disk segment is empty, the stale blocks bypass the disk dump and are transferred to the invalidation segment, provided the invalidation segment has free space to accommodate the incoming stale blocks (line 7–10). The function evicts a flushed block (line 17–20) in two scenarios: (1) the pool of flushed blocks are initially empty, and the stale blocks, not being bypassed the disk segment as stated above, are flushed onto the disk; (2) the pool of flushed blocks are not empty initially (i.e., $\sum_{p=0}^{n_{part}-1} flush_i^p \neq 0$). Function DISKDUMP(i,n), in line 13, dumps n stale blocks of window W_i^{rec} onto the disk. The details of the function is described in 5.3.4. The function UPDATEFREQUENTSEGMENT in line 20 attempts to replace a block within the frequent segment by the evicted flushed block depending on the productivity values of the blocks.

Algorithm 7 AH-EWJ

Input: data streams S_i, window length $W_i (i = 0, 1)$;
Output: results of stream joins

```
1: initialize variables $premature_i$, $freshN_i$, $UI_i$, $flush_i^p$, $stale_i^p$, $BN_i$ {set to 0; i=0,1; $0 \leq p < n_{part}$}
2: loop
3:     retrieve a tuple $s_i$ from input buffer of $S_i$ in FIFO order
4:     $p \leftarrow \mathcal{H}(s_i)$
5:     $W_i^{rec}[p][H] \leftarrow \{W_i^{rec}[p][H] \cup s_i\}$
6:     if $W_i^{rec}[p][H]$ is full then
7:         compute $W_i^{rec}[p][H] \bowtie \left\{W_{\bar{i}}^{mem}[p]\right\}$
8:         $W_{rec(i)}[p][H].BN \leftarrow BN_i$
9:         increment $BN_i$
10:        increment $freshN_i$
11:        if $W_i^{rec}$ is full then
12:            $W_i^{rec}[p] \leftarrow$ EVICTRECENTBLOCK(i)
13:        else
14:            add a free block to the head of $W_i^{rec}[p]$
15:        end if
16:    end if
17:    if $freshN_i \geq rN_{rec}^{min}$ then
18:        $UI_i \leftarrow UI_i + freshN_i \times |b_i^m|$
19:        $freshN_i \leftarrow 0${ reset $freshN_i$}
20:        if $UI_i \geq Epoch_i$ then
21:            DISKPROBE($W_i^{rec}$, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, 1)
22:            $UI_i \leftarrow 0$ {reset the count}
23:        else
24:            DISKPROBE($W_i^{rec}$, $W_{\bar{i}}^{disk}$, $W_{\bar{i}}^{freq}$, 0)
25:        end if
26:        for $p \leftarrow 0$ to $n_{part} - 1$ do
27:            $stale_i^p \leftarrow |W_i^{rec}[p]| - flush_i^p$
28:        end for
29:    end if
30: end loop
```

5.3.6 Join Algorithm

Having described the details of the techniques behind the join algorithm, we now present the join algorithm AH-EWJ in Algorithm 7. Within the join algorithm an infinite loop fetches, in each iteration, tuples from an input buffer and joins the fetched tuples with the opposite stream. At each step, the stream having a pending tuple/block with lower timestamp (i.e., the oldest one) is scheduled. A tuple fetched from the buffer is mapped to its partition and accumulated into the block at the head of the respective partition within the Recent segment. If the head-block of a partition p becomes full, it is joined with the memory portion of the partition ($W_i^{mem}[p]$). The filled block at the head of the partition is assigned a unique block number BN from an increasing sequence. The partition p is allocated a new head block (line 11–15). If no free block is available in the recent segment (W_i^{rec}), a victim window is selected; and a block from a selected partition of the the victim window segment is evicted (line 12).

The variable $freshN_i$ tracks the number of fresh blocks in W_i^{rec}. Whenever a recent segment is filled with at least a fraction rN_{rec}^{min} (r is the invovation threshold; $0 < r < 1$) fresh blocks, the disk probe phase in invoked (line 17). Blocks in Frequent segment W_i^{freq} are updated periodically at the end of an epoch $Epoch_i$; and this update phase is merged with a disk-probe. The epoch can be measured either by time or by the number of tuples processed. We use the later in specifying the epoch. A disk-probe with the update phase (in line 21) is invoked if the number of tuples processed (UI_i) since the last update exceeds the epoch length; otherwise, the *disk probing* omits the update phase setting the last parameter to 0 (line 24). At the end of the disk-probe, parameter $stale_i^p$ for each partition p is changed, converting the fresh blocks within the partition to stale ones (line 26–28).

5.4 Experiments

This section describes our methodologies for evaluating the performance of the AH-EWJ algorithm and presents experimental results demonstrating the effectiveness of the proposed algorithm. We begin with an overview of the experimental setup.

5.4.1 Simulation Environment

We evaluated the performance of the prototype implementation using synthetic traces. All the experiments are performed on an Intel 3.4 GHz machine with 1GB of memory. We im-

Parameter	Defaults	Comment
$W_i(i = 1, 2)$	10	Window length (min)
τ	1	slide interval for a Window (min)
λ	1200	Avg. arrival rate(tuples/sec)
α	0.4	Decay parameter for the productivity value
b	0.6	burstiness in traces (captured by *b-model*)
ρ	0.4	block eviction threshold
M	20	join memory (MB)
N_{flush}	$0.4M$	flush size for RPWJ (MB)
n_{part}	60	hash partitions or buckets
r	0.9	Disk probe invocation threashold

Table 5.2: Default values used in experiments (AH-EWJ)

plemented the prototype in Java. As a buffer, we allocated 2MB of memory and divided this memory between the streams. We incorporated the disk as a secondary storage while buffering the arriving tuples [103]. We generate the streams using the technique described in Chapter 3.

System parameters, metrics and default values

To model the access time for the disk segment, we divide a disk segment (W_i^{disk}) into n_i basic windows $B_{ij}^{win}(j = 1 \dots n_i)$ [52]. We assume that disk blocks within a basic window are physically contiguous, and the delay in accessing the blocks in a basic window is estimated only by the disk bandwidth(i.e., no seek or latency). However, accessing blocks beyond the boundary of a basic window imparts an extra delay equivalent to the seek time and rotational latency (i.e., an access time). We set the size of the basic window to 1MB. We allocate 70% of memory to the recent segments. The remaining memory is equally distributed among the invalidation and frequent segments. Minimum size of a recent segment (B_{rec}^{min}) is set to a fraction 0.3 of the total memory reserved for the recent segments. We set the minimum delay between successive reactive stages (both for RPWJ and AH-EWJ) as 15sec. Note that, for RPWJ, the reactive stages occur at a partition level, whereas AH-EWJ carries out reactive stage (i.e., disk probe while CPU is idle) at a stream level. The flush size (N_{Flush}^{ah-ewj}) for the AH-EWJ algorithm is taken as one basic window. As a base device, we consider IBM 9LZX and use its parameters in measuring the access time.

In our experiments, we fix the memory page and also the disk block size to 1KB. Each record has a length of 64 bytes. The domain of the join attribute A is taken as integers within the range $[0 \dots 10 \times 10^6]$. We allocate 2MB of memory to buffer the stream tuples generated up to a particular instant. We divide the memory between the buffers in the two streams.

When a buffer overflows, we dump the incoming tuples onto the disk. Hence, the total number of the buffered tuples indicates the degree of overload of the system. In normal cases, the buffer size or the volume of pending tuples increases within a bursty interval and decreases while the load is not very high. But, when the system in permanently overloaded, the volume of buffered tuples increases with time saturating the whole system. In addition to the total pending or buffered tuples, we measure *average production delay*, the percentage of *delayed tuples*, total CPU time, total disk time and maximum total window size. This metrics (i.e., average production delay) and the percentage of *delayed tuples* are described in Section 3.5 of Chapter 3. Unless otherwise stated, the default values used in the experiments are as given in Table 5.2.

5.4.2 Experimental Results

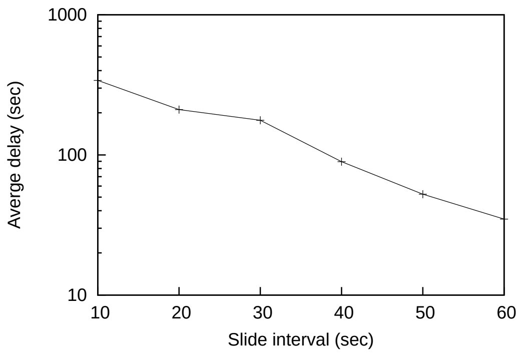
In this section, we present a series of experimental results for assessing the effectiveness of the proposed join algorithm. We measure total number of buffered tuples from both of the joining streams, average delay in generating an output tuple, maximum size of the stream windows, and percent of delayed tuples. For each set of the experimentations, we run the simulator for 1.6 simulation hours. We start to gather performance data after an startup interval of 12 minutes is elapsed.

First we present the experimental results showing the sensitivity of RPWJ algorithm towards two system parameters: the slide interval (τ) and the flush size N_{flush}. Figure 5.4 shows the average delay, the disk-I/O time and the percentage of delayed tuples with varying sliding intervals. As shown in Figure 5.4(a), increasing the slide interval decreases the average delay in output tuples. Increasing the slide interval also results in reduced disk-I/O time (Figure 5.4(b)) and the percent of tuples missing the time limit set by the window (Figure 5.4(c)).

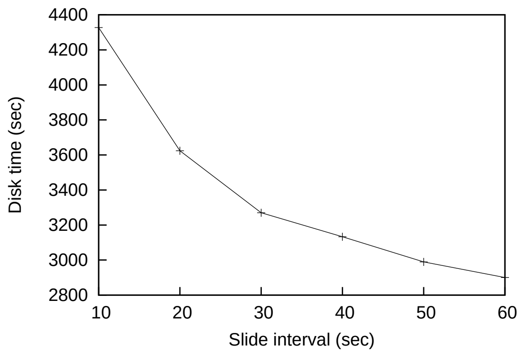
On the other hand, effects of varying flush size on average delay, delayed tuples and the disk-I/O time are shown Figure 5.5. As shown in Figure 5.5(a), the average output delay is minimal when the value of N_{flush} is around the half of the join memory size. So, we we chose a fraction of 0.4 of the join memory as the value of N_{flush} for the subsequent experimentations.

Varying Arrival Rates

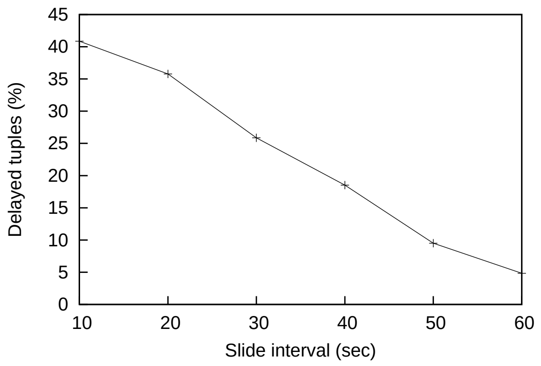
We, now, present the experimental results with varying stream arrival rates. Figure 5.6 and Figure 5.7 show the total buffered tuples (from both the joining streams) at different time-points for the algorithm AH-EWJ and RPWJ, respectively. As shown in the figures, the number



(a) Average delay

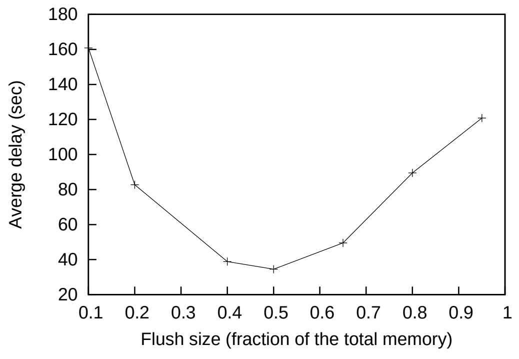


(b) Disk-I/O time

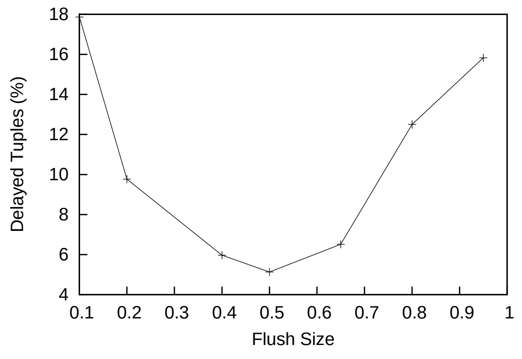


(c) Percentage of delayed tuples

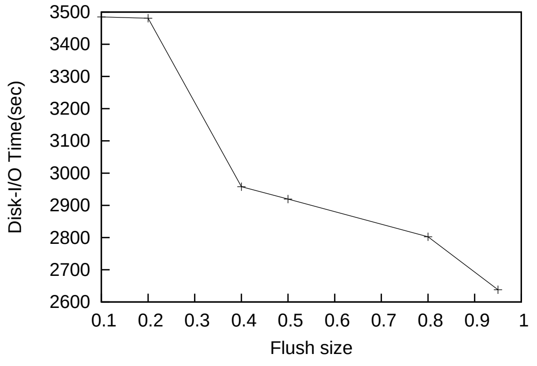
Figure 5.4: Different metrics with varying sliding intervals



(a) Average production delay



(b) Percentage of delayed tuples



(c) total disk-I/O time

Figure 5.5: Effect of varying flush size (expressed as a fraction of the total memory)

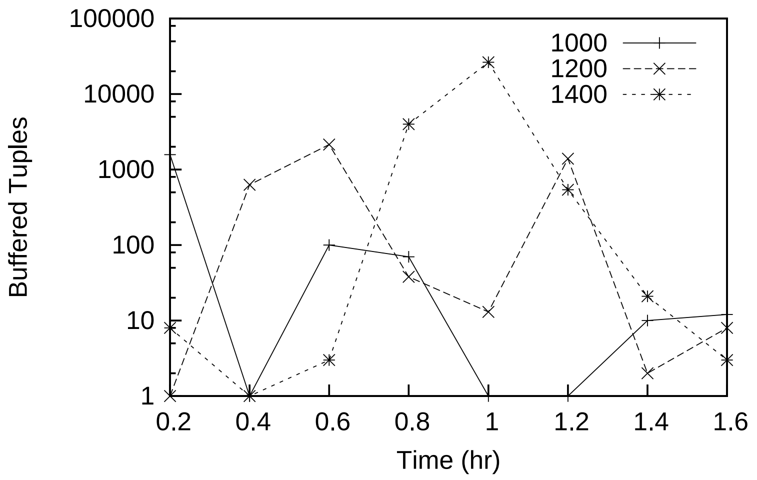


Figure 5.6: Buffer size (for different stream arrival rates) at different time points (AH-EWJ)

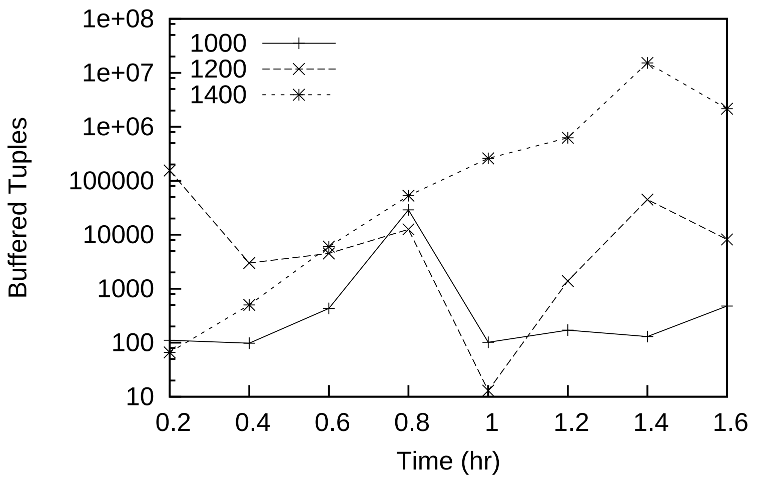


Figure 5.7: Buffer size (for different stream arrival rates) at different time points (RPWJ)

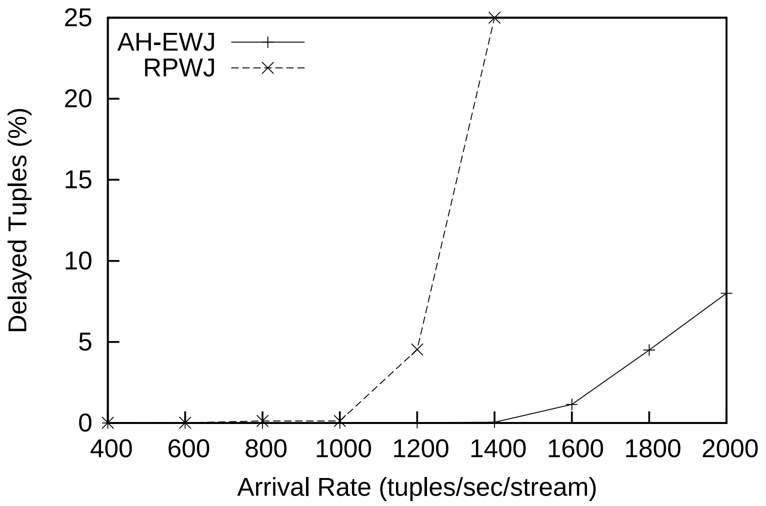
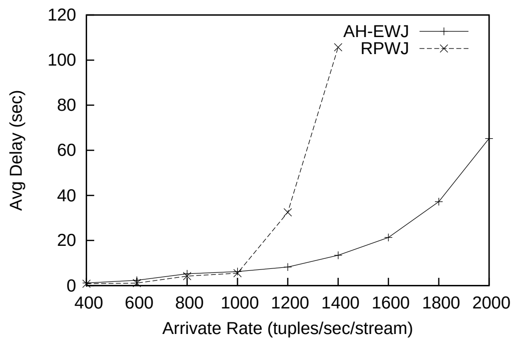
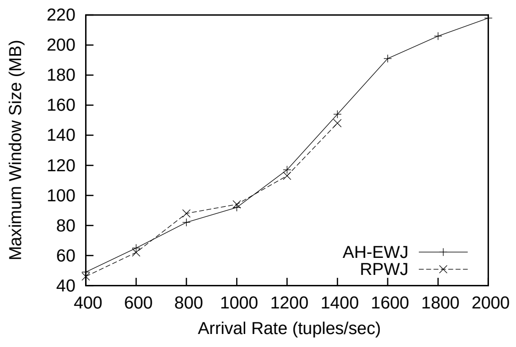


Figure 5.8: Percentage of delayed tuples with varying stream arrival rates



(a) Average production delay



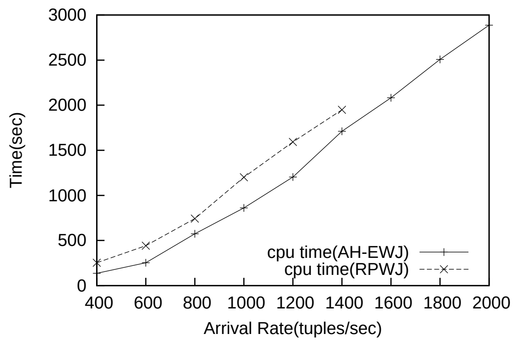
(b) Maximum window size

Figure 5.9: Average delay and maximum window size with varying stream arrival rates

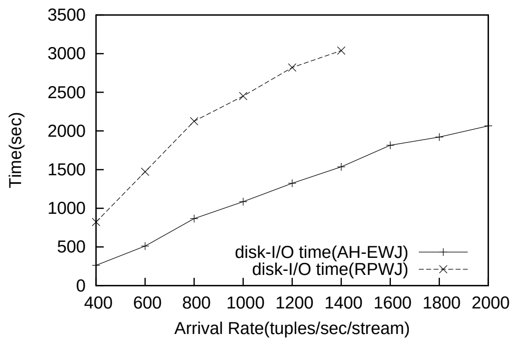
of the buffered tuples increases with the increase in arrival rates. In case of the RPWJ, the system is overloaded for a per-stream arrival rate of 1400 tuples/sec.

With the increase in the arrival rates, more and more tuples can not be processed within the time limit of the stream window; thus, the percent of delayed tuples increases with the increase in the arrival rates as shown in Figure 5.8. The tuples that cannot be processed within the time limit of the stream window is denoted as *delayed tuples*. These delayed tuples get expired at a time later than the usual after they get joined with the respective window. For RPWJ, the percentage of the delayed tuples increases sharply with the increase in arrival rates beyond 1000 tuples/sec; however, in case of the AH-EWJ, this percent of delayed tuples remains low even for an arrival rate of 1800 tuples/sec/stream (i.e., 3600 tuples/sec in the system). Here, it should be noted that *the load applied to the join module is a function of both the window size and the arrival rate; therefore, mere the arrival rate by itself provide little indication regarding the system load.*

Figure 5.9 shows the effect of varying stream rates on the average production delay and the maximum window size. Figure 5.9(a) shows, for both the algorithms, the average delay in producing output tuples with the increase in arrival rates. Figure 5.9(b) shows the maximum window size during the system activity. Though the allocated memory per stream window is 20MB, spilling the extra blocks onto the disk does not impart significant increase in average output delay of the AH-EWJ even for arrival rates up to 1600 tuples/sec/stream, at a point where maximum total window size is around 190MB (i.e., 9 times the join memory size). Techniques based on load-shedding would have discarded the extra blocks beyond 20MB losing a significant amount of output tuples that would never have been generated. The RPWJ algorithm becomes saturated for an arrival rate 1400 tuples/sec/stream at a point where the average delay attains a very high value. Hence, for a large window, the proposed technique



(a) Total CPU time



(b) Total disk-I/O time

Figure 5.10: Total CPU and disk-I/O time with varying stream arrival rates

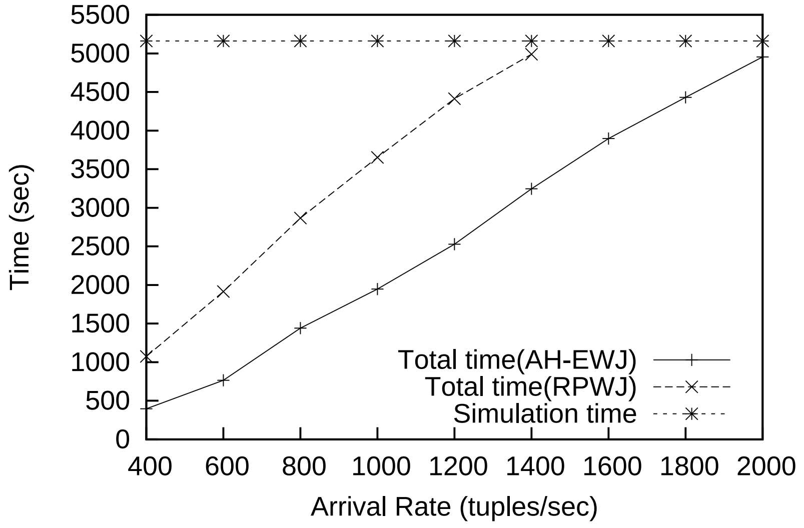


Figure 5.11: Total time with varying stream arrival rates

attains a low average delay of output generation; at the same time, the percentage of the tuples missing the time frame set by the window is very low (0.05% for an arrival rate 1400 tuples/sec/stream). This demonstrates the effectiveness of the AH-EWJ algorithm.

Figure 5.10 shows the CPU and disk-I/O time with varying stream arrival rates. As shown in Figure 5.10(a), the CPU time for the RPWJ is higher while compared to the AH-EWJ. This difference in CPU times can be attributed to the frequent invocation of the reactive stages that do not consider the amortization of a partition-scan over a large number of pending tuples. The disk overhead of the RPWJ, in Figure 5.10(b), is very high while compared to that of AH-EWJ algorithm. In RPWJ the reactive stages are invoked for each individual partition, an approach that results in expensive disk I/Os for a small number of input tuples. Figure 5.11 shows the

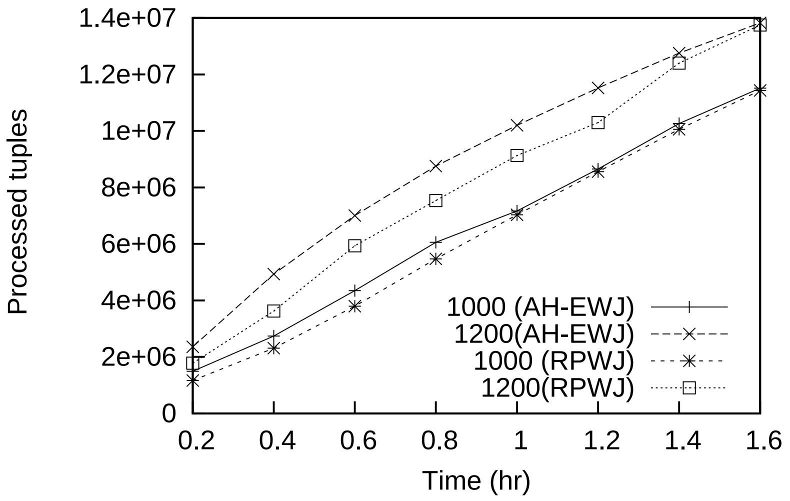


Figure 5.12: Total processed tuples at various time points

total system time (disk I/O and CPU time) with varying per stream arrival rates. We notice that with the RPWJ algorithm the system is saturated at the rate of 1400 tuples/sec/stream. On the other hand, with the same arrival rate, the AH-EWJ algorithms achieves around 40% reduction in total execution time.

Figure 5.12 shows the processed tuples at difference time points. For a fixed arrival rate, the total tuples processed by the two algorithms are same. The pattern of tuple arrivals over time is almost similar for both of the algorithms. However, for same arrival rates, the processed tuples of the two algorithms at various time points might deviate due to the variations in the location of bursts in time.

Varying Hash Partitions

Figure 5.13 shows the effect of increasing hash partitions on the performance of the two algorithms. For RPWJ, increasing the hash partitions beyond a certain values (i.e., 60) results in increased average delay. For RPWJ, as the number of hash partitions increases, the disk overhead also increases. This disk overhead occurs while flushing the tuples, expiring the tuples, and probing disk partitions. As stated earlier, algorithms based on sorting the tuples and merging the sorted *runs* can not be applied in processing sliding window joins. So, increasing the number of hash partitions can be approach to lower the processing overhead. But, the performance of the RPWJ algorithm deteriorates with the increase in the hash partitions; this phenomenon renders the RPWJ unfitting for the sliding window joins. On the other hand, AH-EWJ attains lower average delay with an increase in the hash partitions.

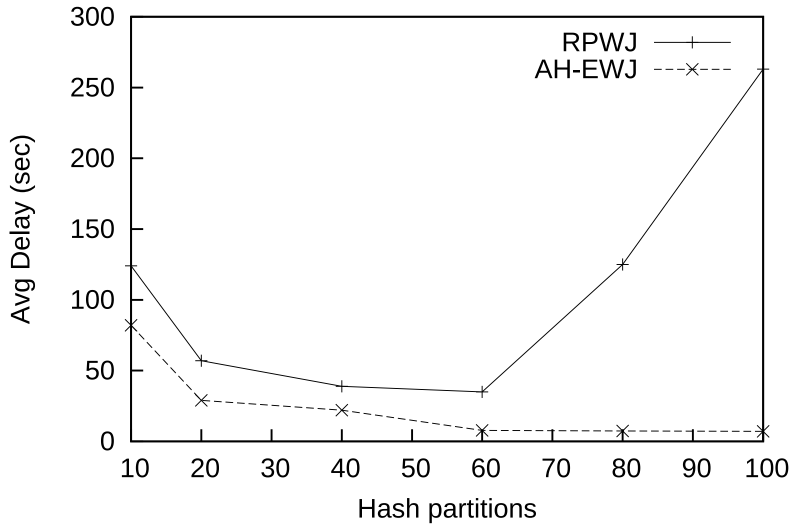
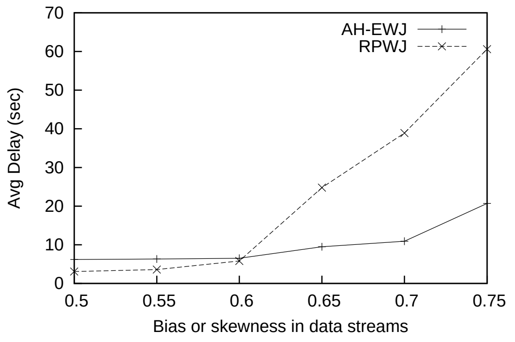
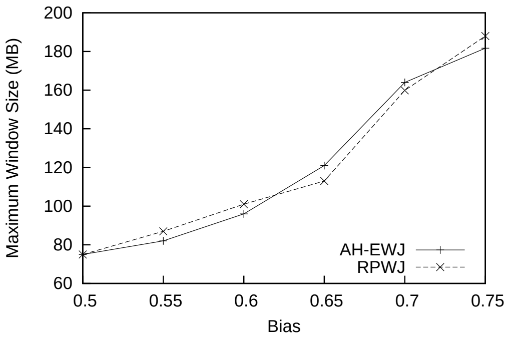


Figure 5.13: Average delay in generating output tuples with varying sizes of hash partitions

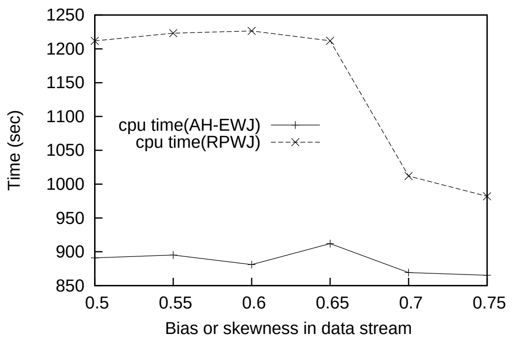


(a) Average production delay

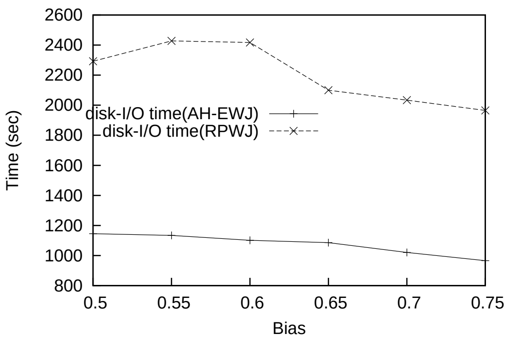


(b) Maximum window size

Figure 5.14: Average delay and maximum window size with varying bias



(a) Total CPU time



(b) Disk-I/O time

Figure 5.15: Total CPU and disk-I/O time with varying bias

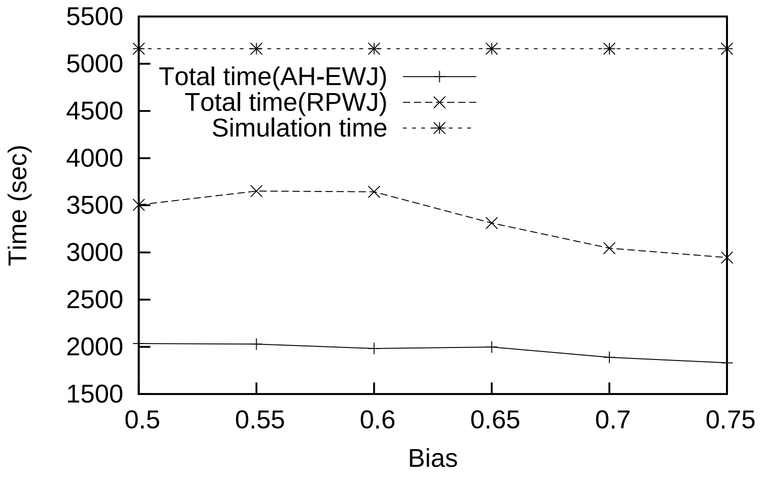


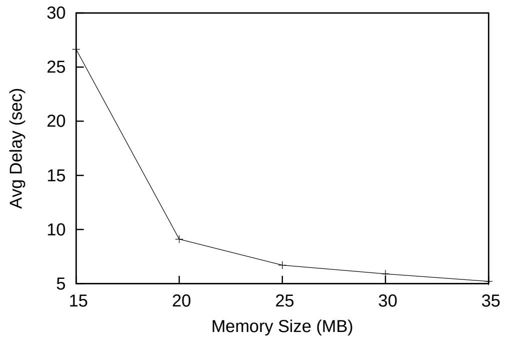
Figure 5.16: Total execution time with varying bias

Varying Bias

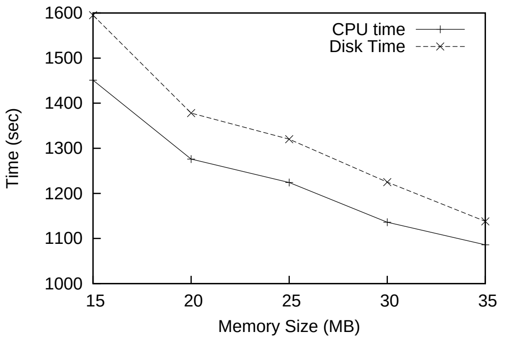
Figure 5.14(a) shows the average delay in generating output tuples with varying bias or burstiness of the data stream. As shown in the figure, with the increase in the bias, AH-EWJ performs significantly better than the RPWJ. Figure 5.14(b) shows the maximum window sizes with varying bias or skewness of stream arrivals. With the increase in bias, the maximum window size also increases.

Figure 5.15(a), Figure 5.15(b), and Figure 5.16 shows the CPU time, disk I/O time, and total System time (disk I/O and CPU time), respectively, with varying skewness in the stream arrivals

Figure 5.15 shows the CPU and disk-I/O time with varying skewness (bias or burstiness) in arrival rates. As shows in Figure 5.15(a) and Figure 5.15(b)), in case of the RPWJ, both the CPU time and disk I/O time decrease with the increase in the skewness. With the increase in the bias, during a bursty interval there occurs, if any, a very few reactive stages. Thus the scan of a stream window partition (while joining the incoming tuples) and disk accesses (to retrieve disk resident stream tuples) are amortized over a larger number of stream tuples. This phenomenon reduces both the processing overheads and the disk I/O time. It is interesting to note that although the maximum window size increases with an increase of the bias, the total processing time, as shown in Figure 5.16, decreases due to the amortization of the both the disk I/O and the window scan over a larger number stream tuples. Such an amortization outperforms the processing overhead due to the increase in the window size.

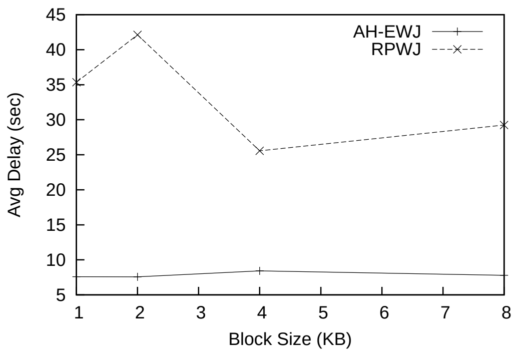


(a) Average production delay

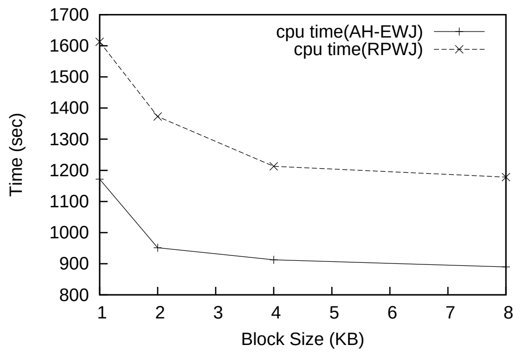


(b) Total CPU time and disk-I/O time

Figure 5.17: Average delay, CPU time, and disk-I/O time with varying memory (AH-EWJ)



(a) Average production delay



(b) Total CPU time

Figure 5.18: Average delay and CPU time with varying block size

Varying Memory Size

Figure 5.17 shows the metrics average production delay, and the CPU and Disk-I/O time with varying available memory for the join operator. Figure 5.17(a) shows, for AH-EWJ algorithm, the average delay in output tuples with varying memory sizes. With the increase in the allocation of memory to process the join operator, the average delay decreases. Also, increasing memory allocation to the join operator decreases both the CPU time and disk I/O time as shown in Figure 5.17(b). With the increase in join memory, the number of blocks spilled onto the disk also decreases, which decreases the disk I/O time during the disk probes. With the decrease in the volume of tuples participating the disk probes, the total processing time also decreases: scanning a window partition during a disk probe requires the partition to be loaded into the cache, increasing the processing overhead.

5.4.3 Varying Block size

The effects of different block sizes on the average delay and the CPU time are shown in Figure 5.18. Figure 5.18(a) shows the average delay in generating output tuples with varying block sizes. In case of the AH-EWJ algorithm, the production delay remains almost fixed, whereas, in case of the RPWJ, the production delay shows some oscillation. However, in either case, there is no significant change in the behavior the algorithms with the change in block sizes.

Figure 5.18(b) shows the CPU time with varying block size. In case of both of the algorithms, the CPU time decreases with the increase in the block size up to a certain. The decrease in the CPU time is due to amortization of the cost of scanning a partition over a larger number of tuples in a block.

5.4.4 Comparison with Relevant Research

In our research, we consider the problem of producing exact results for stream joins given a limited memory. Existing approaches shed load during peak workload by dropping tuples(e.g., [40, 125, 130], and/or processing (e.g., [52]). These approaches fail to produce exact results within an environment where data arrival rate is bursty in nature and/or the length of the stream windows exceeds the available memory. Our scheme handles the burstiness in stream arrivals by spilling data on the disk upon buffer overflow; and it copes with large window by spilling window data on the disk.

The experimental results show that the generation of exact results with bounded average production delay is feasible. We observe, for a window length of 10 min, that the fraction of delayed tuples processed to refine the output results remains very low ($\approx 0.1\%$) even for an arrival rate of 1600 tuples/sec/stream in the system.

5.5 Conclusion

In this chapter, we propose a hash-partitioned algorithm to process the equi-joins in a memory limited environment having burstiness in stream arrivals. Storing the stream windows entirely in memory is infeasible in such an environment. AH-EWJ algorithm proposes a generalized framework to keep highly productive blocks in memory and to maintain the blocks in memory during systems activity. We present techniques to adapt memory allocation across the stream windows at two levels: at a partition level and at a window level. Such an adaption of the sizes of both the windows and the partitions minimizes the disk dumps, reducing the disk overhead. The AH-EWJ provide a technique to spill blocks onto the disk aiming at eliminating random partition switches, which minimizes cache thrashing or the CPU time. The experimental results demonstrate the effectiveness of the algorithm using a baseline algorithm.

Chapter 6

Processing Hybrid Join Queries

In the previous chapters, we provide an algorithm to process stream joins where live tuples arrives in real time. There are some applications (e.g., active data warehousing) that require processing hybrid joins involving one or more live data streams with a single or multiple ersistent tables or relations. Such a join operation is the crux of a number of common transformations (e.g., surrogate key generation, duplicate detection, etc.) in an active data warehouse. In this chapter, we propose a partition-based algorithm to process hybrid joins. The proposed algorithm exploits the spatio-temporal locality within the update stream, and improves the delays in output tuples by exploiting hot-spots in the range or domain of the joining attributes, and at the same time shares the I/O cost of accessing disk data of relation over a volume of tuples from update stream.

6.1 Introduction

In traditional data warehouses, the updates are buffered during busy intervals. The data warehouse is refreshed through the Extraction-Transformation-Loading (ETL) process in an offline fashion (i.e., when the warehouse is idle). This clean separation of querying from updating might render the query results stale. Active data warehouses [22, 80, 140] propagate all updates from the data sources to the warehouse as quickly as possible. Such an online incorporation of updates raises several challenges in implementing the ETL process: transformations should be performed in response to the streaming updates, and the extra task of refreshment should not overload the warehouse. We illustrate the ETL process with the transformation used in surrogate key generation, where keys in update tuples as generated by the data sources are replaced with uniform warehouse keys. This is a join operation between the update tuples

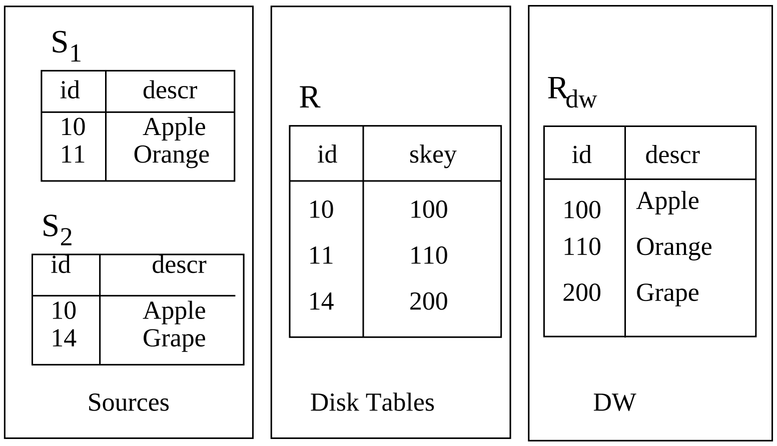


Figure 6.1: Producing surrogate keys

and a look-up table that stores the mapping between the two sets of keys. Figure 6.1 provides an example of the operation where keys (column *id*) from two source tables (S_1 and S_2) are replaced with global and uniform keys (column *skey* in the final table) within the warehouse. In a traditional warehouse, the tuples of S_1 and S_2 are buffered for an indefinite time and the join is performed during idle hours. On the other hand, an active warehouse should perform this operation as the tuples are received from the sources. It should be noted that the inputs to the join operation have different properties: the tuples from the sources arrive at a high rate and should be processed within a time bound; on the other hand, the tuples from the look-up relation are costly to retrieve (from the disk).

The previous example is similar, in nature, to a number of transformations observed in active ETL processes, such as duplicate detection or identification of newly inserted tuples. The operation can formally be defined as $S \bowtie_{\mathcal{P}} R$: here, S is a source update relation, R is a disk resident, warehouse relation, and the join predicate $\mathcal{P}$ defines the semantics of the transformation. In an active warehouse, the join operation should be performed online as the update tuples arrive and thus ensure that the updates are propagated in a timely fashion. Handling the fast arrival rate of stream tuples (or S-tuples) compared to the slow disk I/O (for accessing R) is the major challenge in such a system. Moreover, the join operation should not be resource intensive, since the transformation executes concurrently with other transformations in a pipeline and the transformation should not block the normal system activity. Hence, the operation should use minimal CPU, disk I/O and memory. The combination of the two issues (i.e., mismatch in input access costs and the limited resource requirement) makes the context different from the conventional warehouse architectures. Hence, conventional join algorithms (e.g., hash-join, sort-merge join) are not effective for active data warehousing [115].

The partition-based approach in this chapter joins a fast update stream with a persistent relation. A data stream is shown to exhibit spatio-temporal locality of the stream tuples [91, 95, 137]. This algorithm exploits the locality of update tuples by joining the frequently occurring update tuples within memory partitions. Thus, unlike the MESHJOIN [115], a majority of the arriving tuples might escape even a single disk read, decreasing the average processing time. To decrease the frequency of disk-accesses, we maintain a wait buffer that stores all the incoming tuples corresponding to partitions not maintained in the memory, and invoke a disk probe when the wait buffer is full or the total number of pending tuples corresponding to a partition exceeds a user defined limit called *invocation threshold*. Within the new approach, a stream tuple is served with only one disk read. Unlike the MESHJOIN, this algorithms does not assume a fixed disk relation: the disk relation can be updated during join operations.

This chapter is organized as follows. Section 6.2 defines the problem and discusses the limitations of the existing approaches. Section 6.3 provides the algorithmic details. Section 6.4 presents the experimental studies and Section 6.5 provides a summary of the chapter.

6.2 Preliminaries and problem definition

This section provides an overview of the algorithmic issues we explore. First, we present the problem. Then we provide various approaches to the problem and their limitations. We finish the section after defining the problem in detail.

This chapter considers a class of transformation operations (e.g., surrogate key assignments, duplicate detection, identification of newly inserted tuples) that occur during an ETL process within a data warehouse. These transformations can be mapped to the join operation $S \bowtie_{\mathcal{P}} R$, where, S is a source update relation, R is a disk resident, warehouse relation, and $\mathcal{P}$ is the join predicate dependent on the transformation.

As observed in [115], adaptation of existing algorithms in this setting is infeasible. For example, an Indexed Nested Loop (INL) join, with S and R being the outer and inner inputs respectively, probes the index of the disk relation for each incoming stream tuples (or, S tuple). Thus, this approach suffers large amount of random disk I/Os limiting the arrival rate of the source updates. The limitation of this approach is that it does not amortize a disk access time over a large number of input (S) tuples. One approach to reduce the disk overhead per input (S) tuple is to buffer the arriving tuples from stream S, and invoke the disk scan when the buffer is full. In case of the time varying, bursty arrivals of update tuples, such buffering increases latencies in output results and affects the utilization of the allocated memory.

The MESHJOIN algorithm [115] aspires to reduce the disk access overhead by amortizing a disk access over a large number of stream tuples queued in memory. This algorithm maintains an input buffer, one memory queue (Q) and a hash-table, and divides both the queue and the disk relation into k segments. The memory queue Q stores the tuple pointers, whereas the actual tuples are stored in the hash-table. Each of the disk segment has a size $d = N_R/k$; and, the size of a queue segment $w \approx (M - d)/k$. Here, M is the total memory allotted for the join module, and N_R is the size of the disk relation. To make the equation simple we ignore the memory overhead within the hash table and the space required to store the pointers in the queue Q. Hence, this assumption corresponds to the scenario where the actual tuples are stored in the queue Q and no hash table is maintained [1].

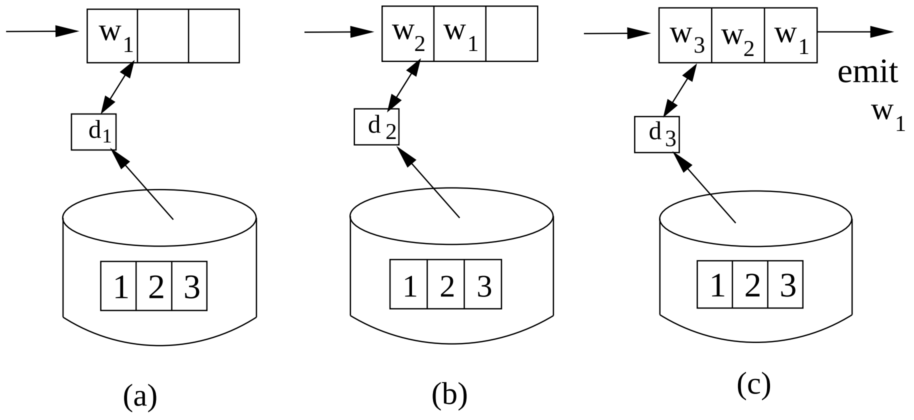


Figure 6.2: Illustration of the operations in MESHJOIN: Each of the memory queue (Q) and the disk relation has 3 ($k = 3$) segments. At $t = t_1$, the first queue segment is full and a disk read retrieves the first disk segment (d_1) of relation R and stores it on the buffer. Now, tuples in d_1 are joined with those in w_1 in memory; at $t = t_2$, when the second queue segment is full, disk segment d_2 is read and joined with w_1 and w_2. After the arrival of the third queue segment w_3(at $t = t_3$), d_3 is joined with queue segments w_1, w_2 and w_3; at this point, w_1 is joined with all the 3 disk segments and is expired from the queue; each subsequent disk invocation joins the tailing queue segment with all the disk segments and results in the expiration of the tailing queue segment.

The incoming stream tuples are stored in the queue segment at the head of the memory queue (i.e., first queue segment). When the queue segment is full, one disk segment is read and joined with the tuples stored in the queue. After the join phase, the last (i.e., tail) queue segment is expired and all the queue segments are shifted toward the tail by one segment. Note that an incoming queue-segment is expired from the memory queue after k disk accesses, a time point when the tuples within the queue-segment are joined with all the tuples in relation

[1]The assumption is made to help realizing the working details of the algorithm; however, our implementation is based on the hash-join

R. Hence, the algorithm carries out continuous scan of the relation R and, on an equilibrium state, finishes processing tuples in a queue segment after each disk access.

Figure 6.2 shows the operation of the algorithm at different time points. Here, each of the disk relation and the memory queue is divided into 3 segments (i.e., $k = 3$). At time t_1, when the queue segment (w_1) on the head becomes full, the algorithm reads the first disk segment (d_1) and joins the tuples within the disk segment d_1 with those in the queue segment w_1 in memory. At this point, the queue segments are shifted toward the tail making the head queue segment empty. At time t_2, the head queue-segment (w_2) becomes full with the incoming tuples. So, the algorithm reads the next disk segment (d_2) and joins d_2 with queue segments w_2 and w_1. At time t_3, the incoming stream tuples fills another queue segment(w_3), and the MESHJOIN algorithm reads the disk segment d_3 and joins with the queue segments w_1, w_2 and w_3. This completes one cycle of scan of the relation R. At this time, the tuples in w_1 are joined with all three segments of relation R. Hence, w_1 can be expired from the queue. Note that, each subsequent disk access would expire one segment from the tail of the queue.

As observed above, a stream tuple is expired after k disk reads, and each disk access is invoked when the incoming stream tuples fill one queue segment. So, in addition to the actual in-memory joining time of the tuples, the delay of an output tuple, measured since the arrival of the stream tuple, are caused by the *disk invocation delay* and the *disk-cycle time*. The former one refers to the fact that an arriving tuple produces no output tuples until a disk probe is invoked, whereas the *disk-cycle time* indicates the total delay in completing one scan of the relation R, which is dependent on k and the distribution of the blocks on the disk. Disk invocation delay can be rendered arbitrarily low by making w very small. This can be achieved by two ways: (1) choosing a very large k and (2) choosing a very large read-buffer size d (which fixes the k-value towards the lower extreme). Increasing the k-value increases the total time to scan the relation R, i.e., the *disk-cycle time*. On the other hand, a large read-buffer size d (which is also equal to the disk segment size) results in a small queue-segment w. However, it decreases the size of the memory queue which is given by $(M - d)$. With a small queue, a costly disk-access (due to high d-value) is amortized over only a small number of tuples, which results in a high disk access time per stream tuple sacrificing the main objective of the MESHJOIN. Also, it is not possible to minimize the disk cycle time below a certain minimal value because of the random delays due to disk head seeks and disk rotations while scanning the disk segments. Such random delays are unavoidable due to the non-sequential storage of the disk blocks of the relation R. Increasing the R-buffer size d only minimizes the disk accesses occurring at the boundary of the disk segments (i.e., inter disk-segment delays). However, it is not possible to minimize the random disk accesses occurring within each disk segment of size d(i.e., intra disk-segment delays). Thus, increasing the disk segment size d, or decreasing k, does not always minimize disk cycle time. [2]

In this paper, we consider the equi-join operation. So, in the join operator, $\mathcal{P}$ is an equality condition over specific attributes of relations S and R. We assume that the disk relation is horizontally partitioned, as in grid files [109], based on the range of the joining attributes. We assume that the amount of the available memory is a small fraction of the size of the disk relation. The (active) warehouse receives a stream of updates S from the data sources in an online fashion. The problem considered in this paper involves (a) the online computation of equi-join between a persistent relation and a update steam with time varying or bursty arrival rates, (b) minimization of latencies observed in the output tuples.

6.3 Partitioned Join

In this section, we introduce the partition-based algorithm for joining a update stream S with a relation R stored on the disk. We start with the overview of the architectural framework and present the detailed features or techniques in subsequent part of the section.

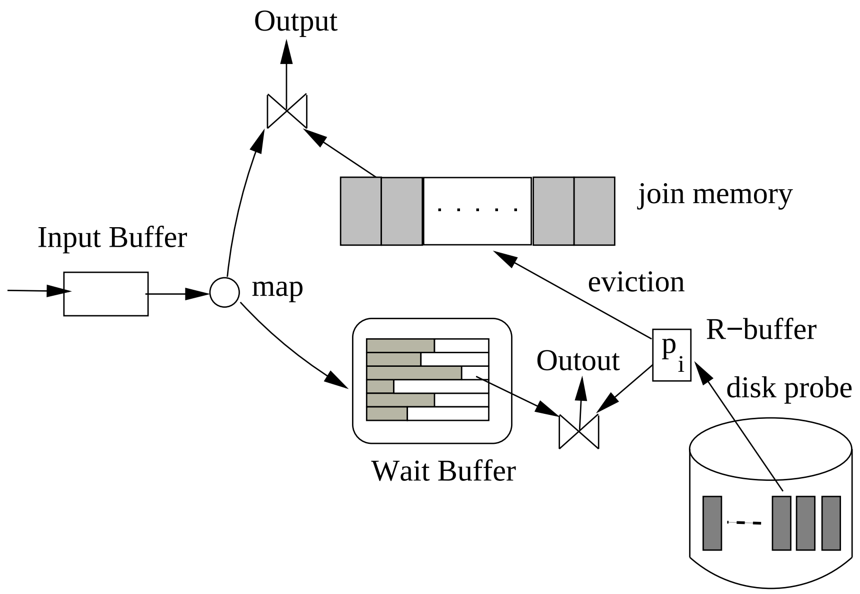


Figure 6.3: The join framework of the Persistent Join

[2]For a given memory budget and a uniform update arrival rate, the optimal performance, for the MESHJOIN, occurs when the read-buffer size d is very small and a major portion of the memory is allotted to the queue Q.

6.3.1 Overview

In the partitioned join, the disk relation is divided into a number of segments by applying a space partitioning (hash- or range-based) technique, as described in the subsequent part of this section, that divides the range of the joining attribute into a number of partitions. A subset of the segments are maintained in the memory by applying a cost-based caching method that attempts to increase the in-memory service rate of the arriving update tuples. An arriving update tuple is mapped to the corresponding partition in the attribute space. If the respective disk segment is available in memory, the update tuple in joined with the disk segment; otherwise, the tuple is stored in a *Wait Buffer*. Within the wait buffer (B_{wait}), which is similar to the parallel buffer [35], a number of buffers (B^i_{wait}) — one for each disk resident segment — share a pool of memory. A *disk probe* is invoked when one of the following condition holds: (a) the number of buffered tuples within a partition crosses a user defined invocation threshold N_{invoke}, or (b) the total space allocated to the wait buffer overflows. The disk probe selects partitions in order of the sizes of the buffered tuples within the wait buffer, retrieves the disk segment (P_i), and joins the disk segment with the buffered tuples corresponding to the disk segment. The retrieved disk segment may replace an in-memory partition depending on the cost-value called *Arrival Frequency* (outlined in 6.3.3) of the partition. If the disk probe phase is invoked due to the buffer overflow (i.e., case (b)) the proposed scheme frees up a certain fraction (r_{wb}) of the wait buffer. Figure 6.3 provides the join framework as outlined above.

6.3.2 Variants of the Algorithms

In our algorithm, the range of the join attribute is divided into a number (n_{part}) of partitions. The disk segment is divided into n_{part} segments based on the partition definition. We consider two types of partitioning, Hash Partitioning and Range Partitioning. In case of the Hash Partitioning, the tuples from both the disk relation and the update stream are mapped, based on hash function, to a fixed number of partitions. On the other hand, Range partitioning defines partitions by splitting the attribute space into a number of contiguous segments. We use the terms range-partitioned join (or, range-based join) and hash-partitioned join (or, hash-based join) while referring to these approaches.

The range-partitioned join algorithm divides the relation into a number of segments, the minimum and maximum segment size (in tuples) being θ and 2θ, respectively. These segments are stored onto the disk, and a data structure defining the partitions is kept in memory which is used to map the incoming stream tuples onto the partitions (or, disk segments). In such a situation, any tree-based structure (binary tree, B-tree etc) for maintaining the partitions

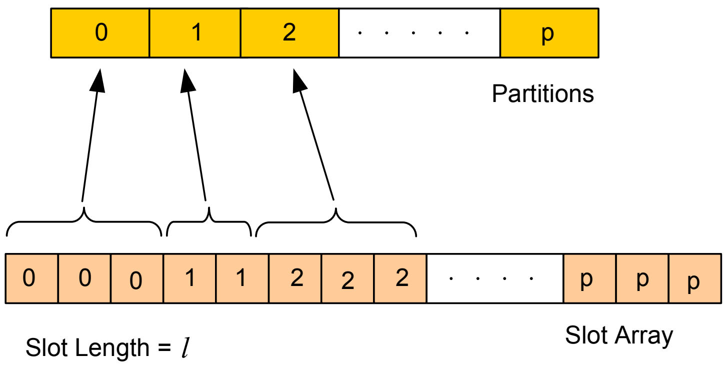


Figure 6.4: Range Partitioning with slot array

would increase the mapping overhead of the update tuples. So, we devise an approach for improving the mapping efficiency. For simplicity, we consider the one dimensional scenario (i.e., single joining attribute A) with non-negative domain (or range) of the attribute. The range of the attribute ($\mathbb{R}$) is divided into 2^n slots so that each slot contains at most θ tuples. Within this Slot-Array, the length of each slot (l) is given by $\frac{|\mathbb{R}|}{2^n}$. The partitioning algorithm clusters the consecutive slots into a number of segments or partitions so that the total tuples within a partition lies within θ and 2θ. Figure 6.4 shows the range partitioning with slot array. Here, each slot entry stores the respective partition or segment number. Given such a slot array, mapping a tuple to the segment is a simple operation. For example, a tuple t_i with A-value a_i maps to a segment given by the entry of the Slot-Array at location $\lfloor \frac{a_i}{l} \rfloor$.

It should be noted that, hash partitioning maps the spatially localized tuples to different partitions or segments eliminating the spatial locality of the stream arrivals. But, in Range-Partitioning, tuples mapped to a partition P_i are bounded by a distance (in attribute space) equal to the span of the partition $n_i l$ (in attribute space), where n_i is the number of slots within the partition P_i.

6.3.3 Maintaining the Memory Partitions

The join algorithm keeps a subset of the disk segments in memory based on a metric, termed *Arrival frequency*, that indicates the volume of recent arrivals of update stream tuples in a particular partition. The arrival frequency (C_k) for a partition P_k ($1 \leq k \leq n_{part}$) is weighted by recency of occurrence using an exponentially decaying scheme [36]. The frequency is maintained over a recent epoch, and the decaying scheme assigns weights to the frequency counts from an epoch according to a negative exponential function of elapsed time. Mathematically,

$$C_k = \sum_{s \in S, s \in P_k} \alpha^{\lfloor \frac{t_{now} - s.t}{\tau} \rfloor}$$

where t_{now} denotes the current time, $s.t$ denotes the arrival time of stream tuple s, and α and τ are user supplied parameters. The parameter α ($0 < \alpha \leq 1$) controls the aggressiveness of exponential decaying. The parameter τ denotes the epoch length(in time). The epoch can also be tuple-based, in which case a new epoch would start after arrival of τ new stream tuples, and $s.t$ and t_{now} would refer to the sequence number of tuple s and the most recent tuple, respectively.

Based on the context of our system, the decaying scheme works as follows: within an epoch, the arrival frequency (C_k) of a partition P_k ($1 \leq k \leq n_{part}$) is increased upon the arrival of a tuple within the respective partition; whereas, at the end of every epoch, the metric C_k is updated as αC_k. Let c_i be the total number of arriving stream tuples within i-th epoch. Now, given the decaying scheme, the arrival frequency immediately after the end of the e-th epoch can be given as,

$$\begin{aligned} C_k^e &= \sum_{i=1}^{e} c_i \alpha^{e-i+1} \\ &= \alpha \sum_{i=1}^{e-1} c_i \alpha^{(e-1)-i+1} + \alpha c_e \\ &= \alpha C_k^{e-1} + c_e \alpha \\ &= \alpha \left(c_e + C_k^{e-1} \right) \end{aligned}$$

The *arrival frequencies* of the partitions are kept in memory as meta data. As stated earlier, a *disk probe* brings a disk segment P_i to the memory and joins with the tuples from the wait buffer. At this point, the retrieved partition P_i evicts a partition P_m from memory if both of the following conditions hold,

$$C_i \geq \frac{|P_i|}{|P_m|} \times (1 + \rho) C_m \tag{6.1}$$

$$M_{free} + |P_m| \geq |P_i| \tag{6.2}$$

In the above equation, ρ is a user defined parameter that controls the aggressiveness of the eviction of a partition within the join-memory, and $|P_i|$ denotes the size of partition P_i. In the first inequality, the arrival frequencies of the two partitions are normalized by their sizes. The

Notations	Description
M_P	set of (disk) partitions/segments in join-memory
B_{wait},B_{in}	wait buffer and input buffer, respectively
B^i_{wait}	set of blocks, in B_{wait}, for partition P_i
B^{free}_{wait}	free blocks in wait-buffer
M_{free}	free blocks join-memory
N_{wait}	total size of the wait buffer
C_i	Arrival frequency metric for partition P_i
N_{invoke}	invocation threshold for *disk probe*
H_{mark} L_{mark}	High and low input buffer occupancy for switching between block- and tuple-level
$B^i_{wait}[H]$	Block at the head of B^i_{wait}

Table 6.1: Notations and the system parameters (Hybrid Joins)

inequality holds if the normalized value for the partition P_i is higher than that of the partition P_m by a fraction ρ. In the second inequality, M_{free} is the number of free blocks available in the join-memory. This inequality simply confirms that, after the replacement, the memory can accommodate the evicting partition.

6.3.4 Join Granularity

Existing approaches to process queries over data streams [30, 40, 125, 53] process incoming streams one tuple at a time. Processing at the granularity of a block might increase the throughput by sharing a scan of the memory segment over a number of tuples within the block. On the other hand, block level processing may impose additional waiting delay. We exploit the information regarding the buffer occupancy and switch the processing granularity between the two levels, block and tuple. When the buffer size is above a high threshold H_{mark}, the algorithm processes at the block level; and when the buffer size is below a low mark L_{mark}, the algorithm switches to the tuple-level. When the current state of the processing is at the block level, the incoming stream tuples corresponding to a partition P_i in memory are stored in the wait buffer until a block is full. When this block is full, it is removed from the wait buffer and joined with the respective disk segment stored in the memory. While switching the granularity of processing to a tuple level, the wait buffer might have tuples corresponding to the partitions maintained in memory. These tuples within the wait buffer are cleaned up removing them from the wait buffer, and joining them with the respective partitions in the memory.

Algorithm 8 PARTITIONEDJOIN

Input: an update stream S, and a persistent table R
Output: Results of the join between S and R

```
loop
    if $|B_{in}| \leq 0$ and $|B_{wait}^{max}| \geq N_{invoke}$ then
        diskProbe($P_{max}$)
    else if $|B_{in}| \leq 0$ then
        {buffer is empty}
        wait for tuples
    end if
    retrieve a tuple $s$ from input buffer $B_{in}$
    $P_i \leftarrow$ mapToPartition($s$)
    if $mode$ = TUPLE then
        if $P_i \in M_P$ then
            compute $s \bowtie P_i$
        else
            $B_{wait}^{i}[H] \leftarrow B_{wait}^{i}[H] \cup s$
        end if
    else
        $B_{wait}^{i}[H] \leftarrow B_{wait}^{i}[H] \cup s$
        if $B_{wait}^{i}[H]$ is full and $P_i \in M_P$ then
            compute $B_{wait}^{i}[H] \bowtie P_i$
        else if $B_{wait}^{i}[H]$ is full then
            select $b_k \in B_{wait}^{free}$
            $B_{wait}^{i} \leftarrow B_{wait}^{i} \cup b_k$
            $B_{wait}^{free} \leftarrow B_{wait}^{free} - b_k$
            $B_{wait}^{i}[H] \leftarrow b_k$
        end if
    end if
    if $B_{wait}$ is full then
        while $|B_{wait}^{free}| \leq r_{wb} \times N_{wait}$ do
            $max \leftarrow \text{argmax}_i\, B_{wait}^{i}$
            diskProbe($P_{max}$)
        end while
    end if
    if $mode$ = BLOCK and $|B_{in}| < L_{mark}$ then
        clean-up the wait buffer
        $mode \leftarrow$ TUPLE
    else if $mode$ =TUPLE and $|B_{in}| > H_{mark}$ then
        $mode \leftarrow$ BLOCK
    end if
end loop
```

Algorithm 9 DISKPROBE(P_i)

```
Input: a partition $P_i$
Output: join results between tuples in the wait buffer and the disk (belonging to partition $P_i$)
1: read disk segment $P_i$
2: compute $B^i_{wait} \bowtie P_i$
3: $|B^{free}_{wait}| \leftarrow |B^{free}_{wait}| + |B^i_{wait}|$
4: if $\exists_m (C_i \geq \frac{|P_i|}{|P_m|}(1+\rho)C_m \wedge M_{free} + |P_m| \geq |P_i|)$ then
5:     $P_i$ evicts $P_m$ from memory
6:     $M_{free} \leftarrow M_{free} + |P_m| - |P_i|$
7: end if
```

6.3.5 Join Algorithm

Having described the details of the partitioned-join algorithm, we now present the algorithm in Algorithm 8. Within the PARTITIONEDJOIN, upon availability of stream tuples, an infinite loop fetches (line 8) the tuples from the input buffer. Line 9 maps the fetched tuples to the respective partition P_i using the procedure MAPTOPARTITION(), that is outlined in subsection 6.3.2. In case of the tuple mode, if the partition, that a fetched tuple s maps to, reside in join memory (i.e., $s \in M_P$), the tuple gets joined with the partition in memory. Otherwise, the tuple is stored in the wait buffer (line 14). On the other hand, in case of the block mode of processing, the fetched tuple s is stored on the block at the head of B^i_{wait} — a list of blocks, in wait buffer, corresponding to partition P_i. If the block at the head ($B^i_{wait}[H]$) is full, one of the following two operations are performed based on the availability of the partition P_i: (a) the block is joined (line 19) with the respective partition P_i of relation R (if the partition P_i is stored in join memory) or (b) a new block (from the pool of free blocks B^{free}_{wait} within the wait buffer) is added at the head of the list B^i_{wait} (line 21–24).

When the wait buffer is full (i.e., $|B^{free}_{wait}| = 0$), a number of disk partitions are probed invoking the procedure DISKPROBE. Partitions are selected in the descending order of the respective block occupancies in wait buffer until a user defined fraction (r_{wb}) of the total size of the wait buffer (N_{wait}) becomes free (line 28–31). Note that, the DISKPROBE is also invoked (in Line 2) when the input buffer is empty and the maximum pending tuples, within the wait buffer, corresponding to a partition ($|B^{max}_{wait}|$) exceeds the invocation threshold N_{invoke}. Such an invocation of disk probing utilizes the idle processor cycles, and also minimizes the processing delay as observed by the pending tuples within the wait buffer. Line 33–38 determine the granularity of processing (i.e., block- or tuple-level) as outlined in subsection 6.3.4. Line 34 cleans up the wait buffer by removing, from the wait buffer, the stream tuples corresponding to the disk-partitions maintained the memory. The tuples removed from the wait buffer are joined with the respective partitions maintained in the join memory. To keep the algorithm simple, we omit the subtle details of maintaining the arrival frequencies C_i which is described in subsection 6.3.3.

Within the DISKPROBE, as given in Algorithm 9, the disk partition P_i is read from the disk and joined with pending tuples in wait buffer. The newly read partition P_i may evict a partition already in join memory depending on the arrival frequency and size of the partition (line 4–7).

6.4 Experiments

This section describes our methodologies for evaluating the performance of the join algorithms and presents experimental results demonstrating the effectiveness of the proposed algorithm. We begin with an overview of the experimental setup.

6.4.1 Simulation Environment

We evaluated the performance of our prototype implementation using synthetic traces. All the experiments are performed on an Intel 3.4 GHz machine with 1GB of memory. We implemented the prototype in Java. We assume that the total memory available to the join operator is very small compared to the size of disk relation R. In case of the partitioned-join, the available memory is allocated to the wait buffer, the disk-read buffer or R-buffer (to store an incoming disk segment) and the join memory; whereas, in case of the MESHJOIN, the memory is allocated to the R-buffer and memory queue Q. As an input buffer, we allocate 1 MB of memory. We incorporate the disk as a secondary storage while buffering the arriving tuples [103]. The number of the buffered tuples indicates when the system is overloaded or saturated; in such a scenario, the number of buffered tuples increases with time. As we incorporate the disk storage with the input buffer, size of the buffer is virtually unbounded. We do not measure the buffering delay as our main objective is to observe the throughput and performance of processing the joins.

Traces

We generate a time varying data stream using two existing algorithms: PQRS [137] and *b-model* [136]. PQRS algorithm models the spatio-temporal correlation in accessing disk blocks whereas the b-model captures the burstiness in time sequence data. In the experiments, we

Parameter	Defaults	Comment
λ	1000	Avg. arrival rate(tuples/sec)
ρ	0.25	Threshold fraction to evict a partition in join memory
α	0.3	Decay parameter for the metric C_i
r_{wb}	0.4	Threshold to invoke disk-probe
N_{invoke}	3000	disk-probe invocation threshold
b	0.6	burstiness in traces (captured by *b-model*)
M	40	total memory (MB)
d	1	read-buffer size(MB) for MESHJOIN)
n_{bucket}	89	hash-table buckets (MESHJOIN and hash-partitioned join)
s	64	size of a S-tuple
r	128	size of a R-tuple
$\|R\|$	200	size of the relation R (MB)
θ	1	minimum segment-size of relation R (MB)

Table 6.2: Default values used in experiments (Hybrid Joins)

generate data streams for a certain duration T. We observe that the time to generate a large volume of data tuples within T while using the PQRS algorithm is prohibitively large as it requires us to sort the tuples based on their times-tamps. So, we divide the duration T into 2^n mini-intervals, measure the total volume of tuples within each mini-interval using the *b-model*. We choose the aggregation level (n) in *b-model* so as to bound the maximum tuple volume within a mini-interval to a given value: this upper limit varies with the parameter b in *b-model* (e.g., for b=0.6 and T = 0.8 hours, we set the upper limit to 70000 tuples). Having determined the total volume of tuples within a mini-interval, these tuples are generated using the PQRS algorithm. We also use this technique to generate the disk relation R. We divide the relation into segments/partitions of length within 1–2 MB.

System parameters, metrics and default values

To model the access time for a disk partition, we divide a disk partition into basic windows [52]. We assume that disk blocks within a basic window are physically contiguous, and the delay in accessing the blocks in a basic window is estimated only by the disk bandwidth(i.e., no seek or latency). However, accessing blocks beyond the boundary of a basic window imparts an extra delay equivalent to the seek time and rotational latency (i.e., an access time). As a base device, we consider IBM 9LZX and use its parameters in measuring the access time. In our experiments, we fix the memory page and also the disk block size to 4KB.

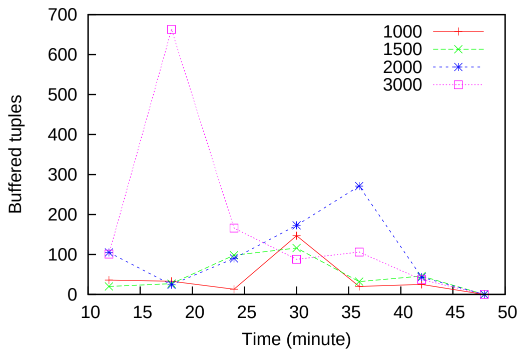
Each tuple from the disk relation R and the update stream S has a length of 128 bytes and 64 bytes, respectively. The domain of the join attribute A is taken as integers within the range $[0 \ldots 10^6]$. We consider a disk relation of size 200MB. For processing the join, we consider a total allocated memory of size 40MB, a memory lower than the size of the disk relation by an order of magnitude. As a wait buffer, in case of partition-based join processing, we allocate a memory pool of size 4MB out of the total join memory(40MB). To support processing join at multiple granularity (i.e., tuple and block) we set the threshold values — high mark H_{mark} and low mark H_{mark}—as 100 and 2, respectively; here, sizes are expressed in blocks. We consider a basic window size of 2MB.

We evaluate the performance of a join algorithm based on a few parameters (e.g.,production delay, disk-I/O time) as outlined in the subsequent part of the section. We measure the number of buffered tuples as an indication of the throughput or overloading of the system. In a normal scenario, the buffer size or the volume of pending tuples increases within a bursty interval and decreases while the load is not very high. But, when the system in permanently overloaded, the volume of buffered tuples increases with time saturating the whole system.

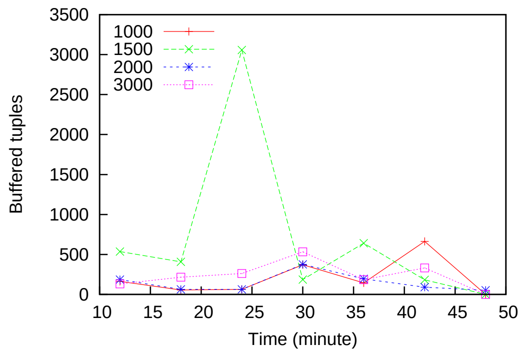
In addition to the total pending or buffered tuples, we measure *average production delay* and total disk time. We measure the delay in producing an output tuple as the interval elapsed since the arrival of the stream tuple. For example, if tuple s from update stream joins, at time T_{clock}, with a tuple r from the disk relation R, then the delay in producing the output tuple is $(T_{clock} - s.ts)$, where ($s.ts$ is the arrival time stamp of the stream tuple s). This metric (i.e., average production delay) indicates how quick an output tuple is generated; hence, can be considered as an indication of instantaneous output generation rate. Unless otherwise stated, the default values used in the experiments are as given in Table 6.2.

6.4.2 Experimental Results

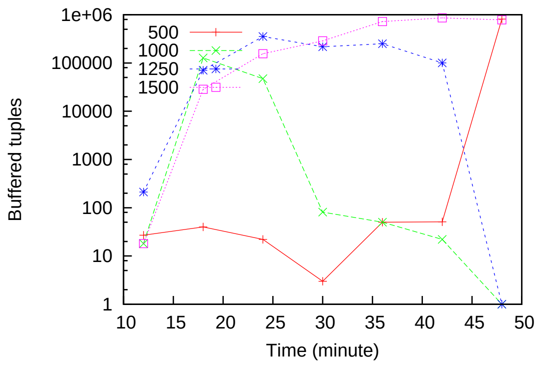
In this section, we present a series of experimental results for assessing the effectiveness of the proposed join algorithm. We measure the input buffer size, average delay in generating an output tuple and total disk time. For each set of experimentation, we run the simulator for 0.8 hour. We start to gather performance data after an startup interval of 6 minutes is elapsed. For the range-partitioned join, the disk relation is divided into a number of partitions of length between 1–2 MB. For the hash-based join and the MESHJOIN, we consider a hash function with the domain $[0..89]$. For MESHJOIN, we set the d-value (i.e., R-buffer size) to 1 MB as the optimal service rate is achieved near this value [115]. As shown in Figure 6.6, the disk overhead is minimal for a R-buffer size within 1-2MB. In case of the partitioned-join (hash-



(a) range-partitioned join



(b) hash-partitioned join



(c) Meshjoin

Figure 6.5: Buffered tuples at different time points with different stream arrival rates

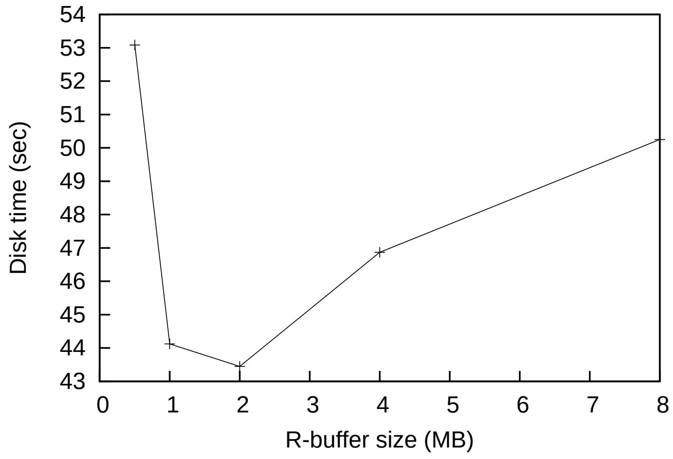


Figure 6.6: disk-I/O time vs rate

based or range-based), we join an arriving stream tuple/block with a segment of relation R using the simple Nested Loop Join algorithm.

We first present the experimental results with varying stream arrival rates. Figure 6.5 shows the total buffered tuples at different time-points. As shown in the figures, the number of the buffered tuples remains very low for partition-based join (Figure 6.5(a) and 6.5(b)). For MESHJOIN, as shown in Figure 6.5(c), the number pending tuples is large even for an arrival rate of 1500 tuples/sec which imply a higher processing overhead and production delay. Within the MESHJOIN, in each cycle, a segment (of length d) is read from the disk and joined with the memory queue using a hash-join. So, the processing overhead can be reduced by increasing the domain of the hash function (i.e., number of buckets in the hash table) used by the hash-join algorithm [3]. However, as the hash-table is maintained for the stream tuples from the memory queue, the table should be purged periodically to remove the tuples expired from the memory queue—an operation that require substantial processing overhead. In MESHJOIN, an arriving tuple should be joined with all the segments (of length d) from the relation R, whereas the partition-based approach limits this scope only within a partition. Also, MESHJOIN works at the granularity of a tuple all the time: the hash table (of the memory queue) is probed for each tuple in the R-buffer. Our proposed scheme, reduces the join overhead by changing the processing granularity depending on the buffer occupancy.

Figure 6.7 shows the average delay in producing output tuples with the increase in arrival rates. This delay for both the hash-based and range-based partitioning approaches is significantly low while compared to that for MESHJOIN (note the log-scale of the graph). This

[3]Applying hash-join within each partition would equally reduce the processing overhead for the partition-based join as proposed. Hence, this issue is orthogonal to the problem we consider

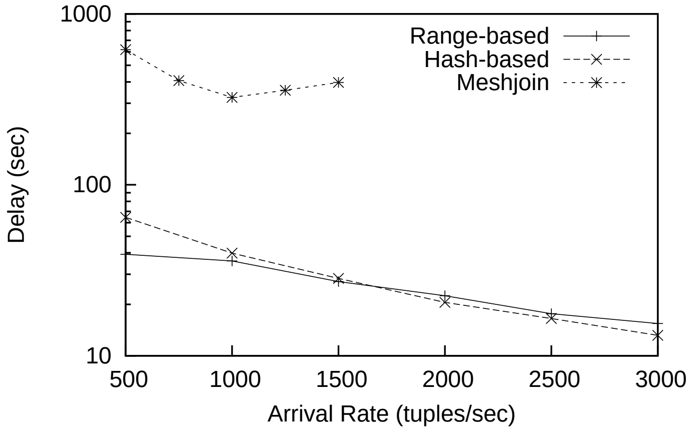
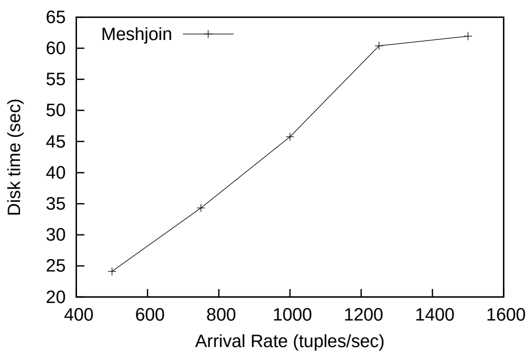
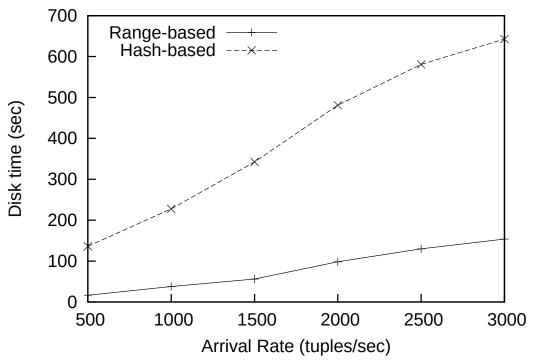


Figure 6.7: Average production delay vs rate



(a) MESHJOIN



(b) Partition-based joins

Figure 6.8: Disk load with varying rates

figure validates the major limitation (i.e., high delay since the arrival until the tuple is fully processed or expired) of the MESHJOIN algorithm as outlined in Section 6.2. In case of the MESHJOIN, as arrival rate increases, up to a certain value the delay decreases. This happens as the queue segments fills up quickly which reduces the disk invocation time. However, for an arrival rate of 1200 tuples/sec and above, the delay increases, as the total processing capacity of the system is used up, and extra tuples are accumulated in the input buffer. Thus the tuples wait in the input buffer before being processed by the system. Such a waiting delay increases the average production delay. This phenomenon is evident in Figure 6.8(a) that shows the total disk access time, for MESHJOIN, during the system activity. Recall that, in a equilibrium state, the total number of disk accesses indicates the total number of tuples processed by the system as a disk-read occur only when a queue-segment is full and a disk read results in the expiration of a queue segment. As observed in the figure, the disk time increases only by a slight margin beyond the arrival rate 1200 tuples/second. This indicates that the extra tuples remain in the input buffer overloading the system.

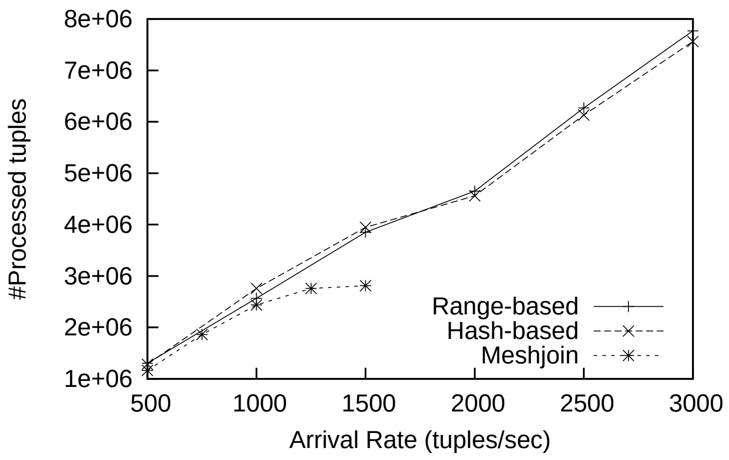


Figure 6.9: Total number of processed tuples with varying rates

Figure 6.8(b) shows the disk load of the two variants of partitioned-join. It should be noted that, for stream rate below 1500 tuples/second, the range-partitioned join algorithm attains lower production delay; however, beyond the rate of 1500 tuples/sec, the delay for the hash-partitioned join is lower with the cost of more disk accesses (cf., Figure 6.7). For the hash-partitioned join, the disk access time is high compared to that of the range-partitioned join, as the spatio-temporal locality of the update tuples are lost after hashing which minimizes the effectiveness of the join memory. For hash-partitioned join, as the arrival rate increases, the pending tuples in the wait buffer crosses the buffer size quickly which causes a disk probe. Moreover, pending tuples for a partition reaches the invocation threshold quickly as tuples are mapped to the partitions in a uniform fashion. This phenomenon decreases the production delay of the hash-partitioned join when arrival rate is high. It should be noted that, the production delay depends on the parameter invocation threshold: decreasing the threshold limit causes frequent disk invocation which reduces the waiting time of the tuples in the wait buffer, hence reducing the production delay. Because of the frequent disk probes, the diskload for hash-partitioned join is larger than that of the range-partitioned join by an order of magnitude as shown in Figure 6.8(b). As evident from the above discussion, at a high arrival rate the average production delay is largely controlled by the *invocation threshold*. At a high arrival rates, the average delay for the for range-partitioned join can be minimized by setting a lower invocation threshold.

Figure 6.9 shows the total number of processed tuples with varying arrival rates. Here, the number of processed tuples do not increase linearly around an arrival rate of 1200 tuples/sec.

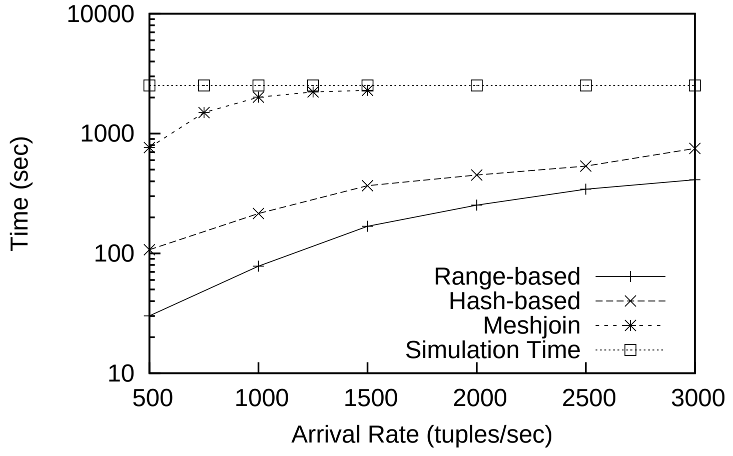


Figure 6.10: CPU time with varying rates

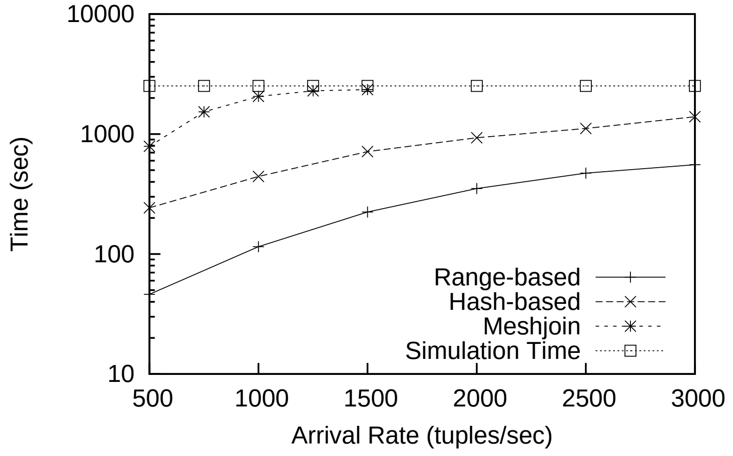


Figure 6.11: Total System time with varying rates

For MESHJOIN, the number of processed tuples remains almost fixed for an arrival rate beyond 1500 tuples/sec. The extra tuples are accumulated at the input buffer, increasing the waiting time at the input buffer. Hence, is such a scenario, for the MESHJOIN, the processing delay of the tuples increases as shown in Figure 6.7.

Figure 6.10 shows, for each of the join algorithms, the total CPU time in processing the join operation. Here, the CPU time is the total time spent by the CPU in actively processing the join operation. The processing time for the MESHJOIN is high, as the MESHJOIN processes the join at the granularity of a tuple, and every tuple is probed against the hash table corresponding to the memory queue. Also, the CPU time for the range-partitioned join is lower than that of the hash-partitioned join. This is because of the fact that the range-partitioned join utilizes the spatio-temporal locality in the stream arrivals, and that achieves lower number of partition switches or disk probes. On the other hand, for hash-partitioned join, partitions are switched randomly, potentially increasing the cache thrashing at L1 and L2 caches. Figure 6.11 shows the total system times with varying arrival rates. The figure shows that, compared to the hash-partitioned join and MESHJOIN, the range-partitioned join significantly decreases the total system time.

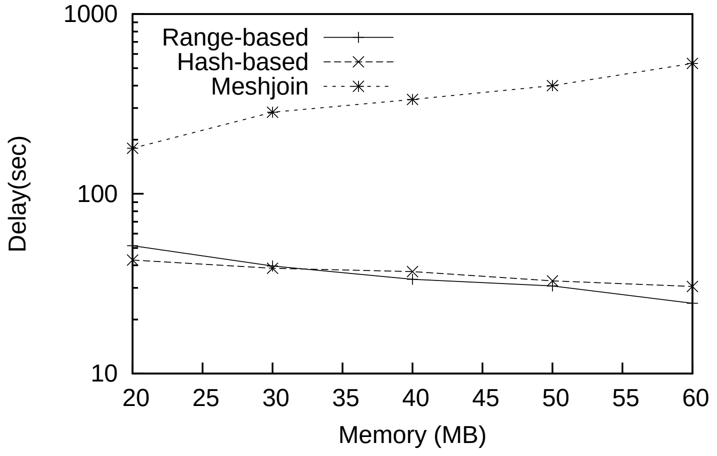


Figure 6.12: Average production delay with varying memory

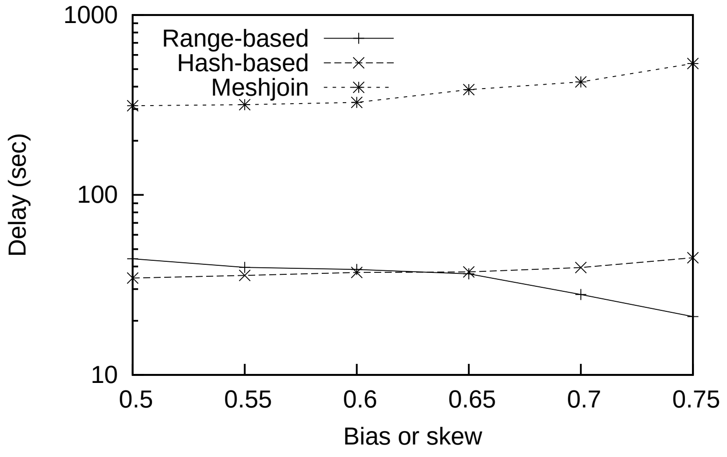


Figure 6.13: Average production delay with varying bias or skew of stream arrivals

Figure 6.12 shows the impact of varying memory on the average production delay. Both the variants of the partition-based approach reduce the delay (note the log-scale) with the increase in the join memory. However, for the MESHJOIN, in line with argument at the beginning of the paper, the delay increases with the increase in the allotted memory, making the algorithm impractical and thus validating our assertion. As the memory increases, the optimal size of the queue-segment also increases, which in turn increases delay between the disk probes.

Figure 6.13 shows the average production delay with varying bias within the stream arrivals. For MESHJOIN, the average production delay increases with the increase in the bias or the burstiness of the stream arrivals. Tuples arriving in bursts overload the system, and are buffered for a long time before they are actually processed, increasing the average production delay. On the other hand, the production delay for the range-partitioned join decreases with the increase in the bias. With an increase in the burstiness, the volume of pending tuples for a partition within the wait buffer reaches the invocation threshold quickly, thus minimizing the waiting time in wait buffer.

6.5 Conclusion

In this chapter, we consider the operation of joining a fast stream of updates with a disk resident relation within a limited memory—an operation commonly encountered in the context of active data warehousing. We identify the limitations of the existing approaches and propose a partition-based scheme with multiple granularity of processing. Processing updates at multiple granularities (tuple and block) achieves significant reduction in processing time. The proposed scheme is I/O efficient, attains high throughput, and reduces processing overheads. We present experimental results validating the applicability and scalability of the proposed scheme to a high stream rate.

Chapter 7

Parallelizing Stream Joins

The stream join algorithms proposed so far considers a single processing node or site. In this chapter, we present an approach to parallelize a stream join operator over a large number of independent processing nodes. We propose a framework, based on fixed or predefined communication pattern, to distribute the join processing loads over the shared-nothing cluster. The fixed or predefined sequence of communication relieves the burden of maintaining any-time, all-to-all communication among the processing nodes, that poses an scalability challenge. We propose mechanisms to adjust the loads across the processing nodes, and to dynamically maintain the degree of declustering in order to optimize both the processing and communication overhead.

7.1 Introduction

A stream join operator is relevant to many applications which need to correlate each incoming tuple with recently arrived tuples from the other streams [52]. Such a window join is used to detect correlations among different data streams, and has many applications in video surveillance, network monitoring, sensor or environmental monitoring. The stream applications place several scalability requirements on the system. First, for high stream rates and large window sizes, a sliding window join might consume large memory to store the tuples of the stream windows [125]. Second, as results need to be computed upon the arrival of incoming data, fast response time and high data throughput are essential. Third, some join queries such as video analysis can be CPU-intensive [66]. Fourth, a typical data stream management system could have numerous window join queries registered by the users. Thus, a single server may not have enough resources to process the join queries over a high stream rate.

Scalable processing of data streams over a distributed system has been studied by the researchers. Reference[145] proposes a dynamic load distribution framework by partitioning the query operators across a number of processing nodes. Thus, this approach provides coarse-grained load balancing with inter-operator parallelism. However, such an inter-operator parallelism do not allow a single operator to collectively use resources on multiple servers. In [67], the authors address the issue of diffusing the join (both equijoins and non-equijoins) processing loads across a number of servers, and provide two tuple routing strategies satisfying *correlation constraints* for preserving join accuracy: *aligned tuple routing* (ATR) and *coordinated tuple routing* (CTR). The former one might achieve poor load-balancing across the nodes, and in the worst case might result in overloading a master node receiving a major part of the processing load (Section 7.3).

In [134], the authors presents an algorithm to parallelize a range join query over a distributed database. In such a system, the data is replicated over a number of nodes, and the algorithm adjusts query result throughput to match with the network bandwith at the users' end. Such an adaptivity is achieved by adjusting the data placement and the number of nodes to be employed to carry out the range join processing. The presence of live data streams (contrary to fixed and stored data sets available beforehand) and state-based join operator render the stream join problem challenging and novel. Instead of adjusting the degree of parallelism to match the result throughput with that of the users' bandwith, we adapt the degree of parallelism to reduce the processing and communication (or, co-ordination) overhead.

In this chapter, we consider the issue of parallelizing a window join over a shared-nothing cluster to achieve gradual scale-out by exploiting a collection of non-dedicated processing nodes. In such an environment, a processing node can be shared by multiple applications; therefore, the need for over-provisioning for the peak load of any application is not necessary. As multiple applications or users share each node, the non-query background load and the available memory for processing queries vary on each of the nodes. Since the continuous stream join queries run indefinitely, the join operator will encounter changes in both system and workload while processing the queries. In such an environment, intra-operator parallelism of a window join can be achieved by partitioning the streams across the processing nodes and instantiating the window joins within every processing node that process the join over a subset of the partitions of the streams. To achieve optimal performance, the system should adjust the data stream partitioning on the fly to balance resource utilization.

We parallelize the window join over a shared nothing cluster by hash-partitioning the input streams, distributing a subset of partitions to the available nodes, and adjusting the dataflow towards the nodes based on the availability of the resources within the nodes. We partition each input stream and create partition-groups by combining partitions with the same partition ID from all input streams; we use this partition-group as the smallest unit of adaptation. Considering the nature of communication primitives (e.g., receiving a packet must block the receiver if the sender is not available) within any persistent or reliable connection, e.g., Transmission Control Protocol (TCP) [126], we propose a framework to distribute the incoming tuples and adapt the loads across the slaves (i.e., the processing nodes). The slaves, instantiating a consumer operator (similar to exchange [64] or Flux [122]), communicate with a master node periodically at the end a *distribution epoch*, and receive stream tuples from the master equipped with a producer operator, that distributes the streams among the slaves. The master node invokes a repartitioning protocol at the end of a *reorganization epoch*: the master node, based on the load information from the slaves, identifies the suppliers and consumers of partition-groups (i.e., processing loads) and notifies the slaves about the state movement. A slave node joins the incoming tuples with the partitions from the opposite streams using a simple nested loop join; other join algorithms based on sorting are not feasible as the temporal order of the tuples should be preserved to allow efficient tuple invalidation.

With the increase in arrival rates, the size of the individual partitions within each partition-group increases. This phenomenon limits the scalability of the join algorithm: as partitions grow in size, the CPU-time to scan the partitions and join with a new tuple also increases. To ameliorate this problem, we fine tune the partition-groups at each processing node by dynamically adjusting the sizes of each partition: we split a partition if the partition size exceeds a predefined *split threshold*, and try to merge a partition with another one from the same stream and the same partition-group if the partition size shrinks below a *merge threshold*. In summary, the key contributions of this chapter are as follows:

1. We propose a technique to support fine-grained, intra-operator parallelism while executing a stream join operator over a shared-nothing cluster. The proposed technique do not assume all-time, any-to-any persistent communication among the participating nodes, eliminating the scalability overhead.

2. We observe a performance bottleneck in processing the window join over high stream rates, and propose a solution methodology based on the fine-tuning of the window partitions locally in each processing node.

3. We analyze the overheads in scaling the system to a large number of nodes and propose the methodologies to optimize the scalability overheads of the system.

4. We implement the algorithm in a real system, and present experimental results showing the effectiveness of the techniques.

The rest of the chapter is organized as follows. Section 7.2 provides the basic concept in processing sliding window joins, and presents the system model considered in the paper. Section 7.3 defines the problem and provides an overview of the proposed algorithm. Section 7.4 describes load balancing technique in details. Section 7.5 considers the issue of scaling the system to a large number of nodes, and proposes techniques to reduce the system overheads (e.g., processing and communication overhead). Section 7.6 presents the experimental studies, and Section 7.7 summarizes the chapter.

7.2 System Architecture

The windowed join operator computes the join results over sliding windows of multiple streams. For a stream S_i, we use r_i to denote the average arrival rate in stream S_i. In a dynamic stream environment, this arrival rate can change over time. Each tuple $s \in S_i$ has a timestamp $s.t$ identifying the arrival time at the system. As in [11], we assume that the tuples within a stream have a global ordering based on the system's clock. We use $S[W_i]$ to denote a sliding window on the stream S_i, where W_i is the window size in time units. Abusing the notation a little, we use W_i to denote the window $S[W_i]$; the difference will be explicit from the context. At any time t, a tuple s belongs to $S_i[W_i]$ if s has arrived on S_i within the interval $[t - W_i, t]$. The output of a sliding window equi-join $S_1[W_1] \bowtie \cdots \bowtie S_n[W_n]$ on a join attribute A consists of all composite tuples $s_1, \ldots, s_n$, such that $\forall s_i \in S_i, \forall s_k \in S_k[W_k]$ $1 \leq k \leq n, k \neq i$ at a time $s_i.t$, and $(s_1.A = \cdots = s_n.A)$.

The distributed stream processing system consists of a cluster of processing nodes connected by a high-speed network. Data streams from various external sources are pushed to a master node that serves as gateway to distribute workload across the slaves as shown in Figure 7.1. The join queries from the users are submitted to the master node. For a given stream join query, the master node selects the number of slaves to instantiate the join operator in. Moreover, the master node stores the incoming stream tuples within a buffer, and periodically sends the tuples to the slaves which carry out the actual processing. The join results from the slaves are routed to a collector node that merges the query results and sends to the respective users. Thus, the shared-nothing stream processing system appears to an user (or client) as a unified stream processing service to serve a large number of continuous windowed join queries over high volume data streams.

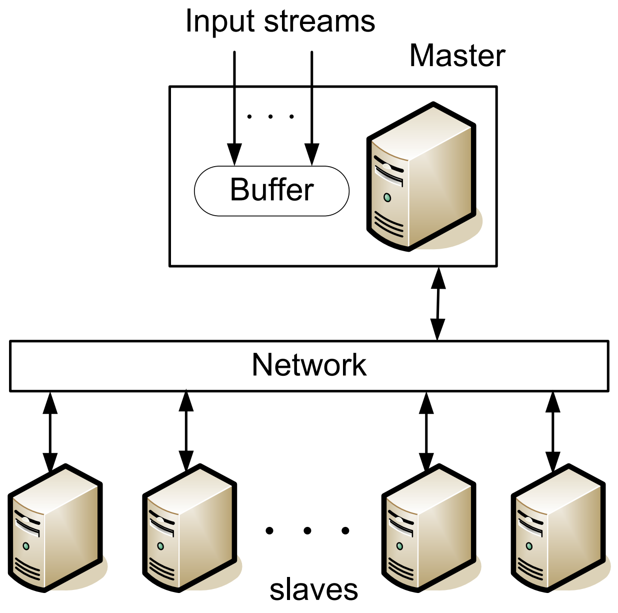


Figure 7.1: The System Model for processing window join

7.3 Problem Definition and Solution Approach

This section presents the problem considered in this paper, and provides various approaches to the problem and their limitations. The section ends with detailed definition of the problem.

This paper considers an operator that joins sliding windows of two streams S_1 and S_2, i.e., W_1 and W_2, respectively. For a join attribute A, we aim at answering the continuous join operator $W_1 \bowtie W_2$ with condition $s_1.A = s_2.A$, for all $s_i \in W_i (i = 1, 2)$. The join operator over the recent windows of the streams are continuously evaluated, at different time points, with the arrival of stream tuples. Tuples in a stream are organized into n_{part} partitions. Thus each window is splitted into n_{part} partitions based on a hash function $\mathcal{H}$, and denote the j-th partitions of a window W_i as $W_i[j]$. Given the partitions of the windows, the hash join can be evaluated by simply joining the tuples within the partitions (from different joining streams) with the same partition ID, as given by

$$W_1 \bowtie W_2 = \cup_{j=1}^{n_{part}} W_1[j] \bowtie W_2[j]$$

In a distributed environment, initiating data exchange or communication by an application at any point in time, and without any prior synchronization or predefined order, is infeasible. For example, an application receiving data over a persistent connection must block if the sender node is not scheduled to send the data. Such a phenomenon degrades the system performance, as an application, while blocked, can not continue to process incoming tuples until the data from the sender is received (i.e., the receiver is unblocked). Applications can be relieved of such blocking by providing multiple helping threads or processes which invoke the receive primitive on behalf of the *main thread* or *process*. The helping threads continuously poll the sender (by invoking a *receive* primitive), store the received data in queue maintained by the main thread; the main thread checks the queue at the start of each iteration and carries out necessary processing with the received data in the queue. Such an approach should maintain multiple helping threads or processes (one for each node in the system). Moreover, the frequent switching of the threads/processes imparts a significant overhead while switching the contexts, thus limiting the scalability of the system. The overhead of managing multiple connections can be minimized by using User Datagram protocol (UDP) [126] for communication. However, UDP is unreliable and results in data loss, which renders the option infeasible in processing data streams.

In reference [67], the authors first address the issue of intra-operator parallelism while processing a join operator over a number of servers. The paper provides two tuple routing strategies, namely *aligned tuple routing* (ATR) and *coordinated tuple routing* (CTR), that preserve join accuracy. The ATR assigns a segment of the master stream to a selected node, and changes the assigned node at the end of every segment. The ATR works for a segment much higher than the sizes of the stream windows. Thus, instead of balancing the loads, this approach circulates the join processing load across the nodes: during a segment interval the node assigned with the segment of the master node carries out all the join processing loads, while the remaining slave nodes (assigned with a segment of the slave streams) only forward the incoming tuples of the slave streams to the respective master node. This approach violates the assumption of resource limited processing nodes; for example, storing the windows of all the streams in a master node (within a segment of the master stream) is infeasible. Selecting a small segment length do not ameliorate the problem; in this case, when a subsequent segment of the master stream is assigned to a new node, all the stream windows should be routed to the new master node, and this happens at the end of every segment. On the other hand, CTR distributes the stream segments across the participating nodes, and maintains a routing path for each stream. The routing path is a sequence of routing hop , and each routing hop V_i is a collection of nodes storing a superset of the i-th stream-window in the join order. In addition to the computational overhead associated with maintaining the routing paths, such an approach incurs high network overhead, as each incoming tuple (from a stream source) or intermediate results (generated by a segment of the intermediate *routing hop*) should be forwarded, in a cascading fashion, to every node in the successive *routing hop*.

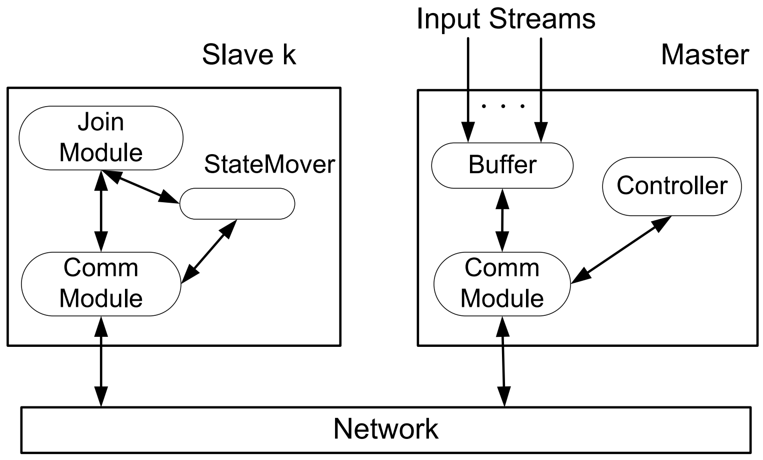


Figure 7.2: System Architecture showing various software components at different types of nodes

We consider the issue of computing sliding window equijoins joins over a shared-nothing cluster forgoing any notion of persistent, any-time, all-to-all communication pattern among the nodes. The algorithm should follow a fixed communication sequence or predefined order of data exchange among the processing nodes. The problem considered in the paper involves (a) dynamically balancing the join processing loads among the slaves, (b) determining the degree of declustering based on the communication overhead, total processing time, and the processing load at the nodes, (c) study the tradeoff between the latencies in the output tuples and communication overhead.

7.4 Join Processing by Synchronous Communication

In this section, we introduce the load diffusion algorithm, based on Synchronization of communication among the participating nodes, to process hash join over a shared-nothing cluster. We start with the architectural framework and present the detailed features of the system in subsequent parts of the section.

7.4.1 Overview

The join processing system consists of two categories of nodes, a master and the slaves, that communicate over a network using communication primitives (i.e., *send* and *receive*) over a reliable and persistent connection. The software components of the nodes are shown in Figure 7.2. The master node stores the incoming stream tuples in a buffer. The streams are divided into partitions (or substreams) based on a hash function. Each slave node is assigned with a subset of the partition-groups (i.e., partition with same partition ID within the streams). The buffer also maintains the mappings between the partition ID and the slave machines. The slaves communicate with the master node periodically , at the end of predefined time intervals, and exchange stream tuples and/or system load information: at the end of every *distribution epoch*, when the slave nodes initiate the communication, the master node sends the buffered tuples to the slave nodes; whereas, at the end of every *reorganization epoch*, the master node, based on the observed workloads at the slave nodes, adjusts (a) the mappings between the partition-groups and the slave nodes, and (b) the degree of declustering, i,e., the number of slaves actively participating in processing the sliding window join. The controller module in the master node carries out the processing for reorganization. On the other hand, a slave node receives stream tuples and special instructions (e.g., move a window partition) from the master, and initiate relevant processing.

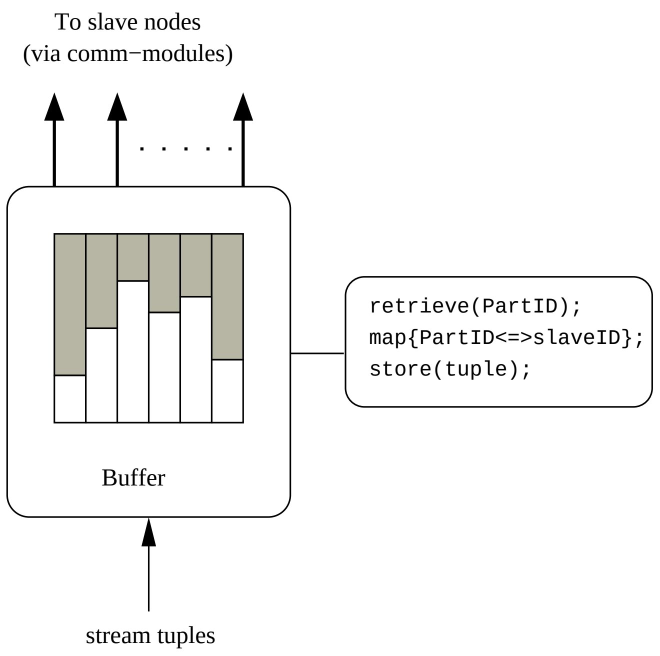


Figure 7.3: Buffer at the master node

7.4.2 Tuple Distribution

A master node maintains, for each stream, a buffer that consists of multiple "mini-buffers", one for each partition (Figure 7.3). The buffer keeps the mapping between the partition ID and the slave machines assigned with the partition. From such a mapping, the partitions assigned to a slave can be identified. At the end of a distribution epoch, when the slave node communicates with the master, the master drains the tuples from the "min-buffers" corresponding to the partitions assigned to each slave, and sends the merged tuples to the slave node in machine independent format. As the tuples from all the streams are merged, the merged stream, received at the client, should have enough information to map the tuples to the respective source streams. Two approaches are possible: augmenting an extra attribute, containing the stream ID, with each stream tuple; and putting special punctuation marks (which might itself be fictitious tuples) at the sequence of tuples from each stream.

On the other hand, a slave node receives the tuples from the master, and stores the tuples in a stream buffer (at the join module). Once the tuple distribution phase is finished, the join module start to process the tuples from the buffer. The buffer maintained at a join module within a slave node is similar to that in the master except for the mapping information.

7.4.3 Repartitioning

The input streams are repartitioned to re-balance the works across the slave nodes. To facilitate such re-balancing of works across the slaves, we introduce a level of indirection while partitioning the input streams: instead of having one giant partition per slave node, we instantiate numerous partitions, so that the total number of partitions is much higher than the maximum degree of declustering of the hash join [122, 41]. These numerous partitions are distributed among the operator instances at the slaves during initialization, and those instances are responsible for processing the corresponding inputs. The repartitioning task requires moving the window states for processing the subsequent input tuples.

The processing nodes in the system (master and the slaves) start the repartitioning protocol periodically at the end of an interval called reorganization epoch, which is a order of magnitude larger than the distribution epoch; a large value of the reorganization epoch is necessary to capture the long term variation in join workloads. In this protocol, the slave nodes send to the master the information about the loads. We use an average buffer occupancy metric, over the current reorganization epoch, at a slave as the indication of the load applied to the slave. A slave node records the buffer size at the end of each distribution epoch within current reorganization interval and calculate their average. The average buffer occupancy metric is obtained by dividing the calculated average buffer size by the total buffer size (i.e., memory allotted to the buffer); we assume that the memory allocated to the buffer in every slave is the same. Based on this average buffer occupancy (f_i) values of the nodes, the master classifies the slaves into one of the three categories: *supplier*, *consumer*, and *neutral*. A *supplier* i has an average buffer occupancy f_i above a threshold value Th_{sup}, whereas a consumer is a slave with the average buffer occupancy below Th_{con} ($0 \leq Th_{con} < Th_{sup} < 1$); the rest of the nodes, which are neither supplier nor consumer, are classified as the *neutrals*. A supplier yields a partition-group to a consumer node, which installs the partition-group in its join module and processes the subsequent tuples arriving within the partition-group. As outlined in section 7.5, in a stable system, the number of consumers is higher than that of suppliers.

While reorganizing the placement of partition-groups across the nodes, each supplier yield only one randomly selected partition-group to a consumer. For each consumer, the master node selects a supplier, and sends messages to the participating nodes to initiate transferring the window states of the partition-groups to move. The supplier-consumer pairs can be identified by a single scan over the list of the slave nodes. The master node makes necessary changes in the mapping between the stream partitions and the slave nodes. To transfer a state, the *state-mover* in a slave node (supplier) extracts the tuples from both the stream windows and the buffer (at the join module), and sends to the consumer. The splitting information, if any, is also sent to the consumer to enable it reconstruct the fine-tuned partitions. After finishing the state movement, the participating nodes send acknowledgement to the master indicating the completion of the task, upon which the master node transfers the pending tuples (in its buffer) to be processed. This completes both the reorganization and the tuple distribution phases. Note that the slaves not participating in the state movement receives the pending tuples before the master receives an acknowledgement from every node participating in the state movement.

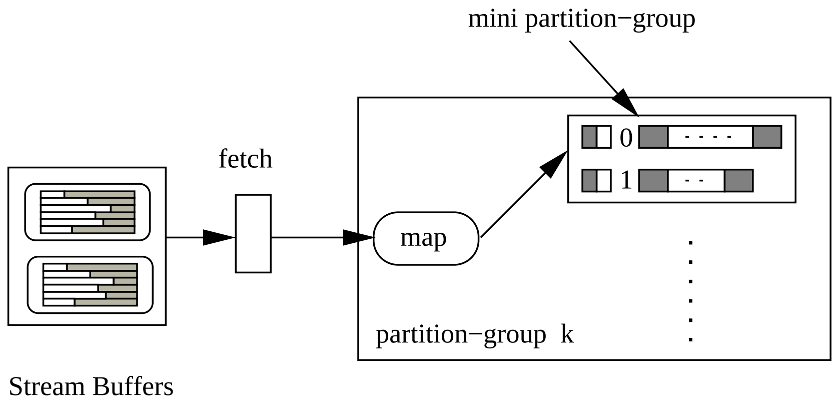


Figure 7.4: Processing tuples at a join module within a slave node

7.4.4 Processing at the Join Module

Each slave node receives stream tuples from the master at the end of every distribution epoch, and stores the tuples in stream buffers. As discussed earlier, each buffer keeps the tuples for each partition in a separate mini-buffer, so that blocks corresponding to a partition can be retrieved from each mini-buffer without scanning the whole buffer for the stream. The join module fetches a block for a partition of a stream, *maps* the tuples to the mini-partition, and keep the tuples in block at the head of the mini window-partition of the stream (Figure 7.4). When the head block is full or the buffer contains no more blocks for the respective stream partition, the newly added tuples are joined with the mini-partitions from the opposite stream windows. Also, as obvious from the above description, the tuples in a head block may participate in the join when it is not full. To utilize the empty portion of the head block, a variable is used to track the newly added (i.e., *fresh*) tuples within a head block. To eliminate duplicate output results, the fresh tuples within the head blocks of the opposite mini window-partitions are omitted, as the head blocks will participate in the join when they become full. While expiring a block from a mini-window, the block is joined with the fresh tuples within the head block of the opposite mini-window; this ensures the completeness of the join results.

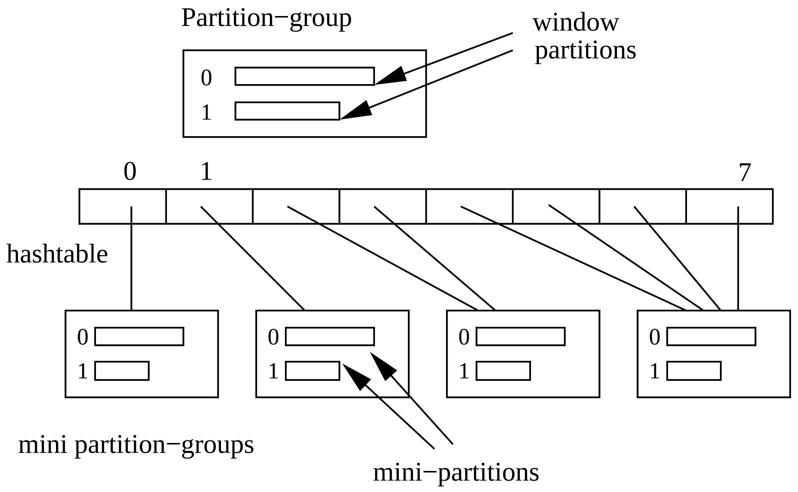


Figure 7.5: Fine tuning the window partitions at a slave node based on extendible hashing

The size of a (mini) partition of a stream may grow and shrink depending on the arrival rates of the tuples within the respective partition. As the size a window partition increases, time needed join the partition with the tuples from the opposite streams also increases. On the other hand, a small partition size increases the memory overhead. Thus we should adjust the size of the partitions. We adapt the size at the granularity of the partition-groups. We keep the size (in blocks) of each partition-group within a range $[\theta \dots 2\theta]$. We split a partition-group when its size is above 2θ, and try to merge partitions whenever a partition size falls below θ. We maintain the size of the partition-groups using extendible hashing [48], and maintain one hashtable for each overflowing partition-group(Figure 7.5). We split a partition-group using the extendible hashing, and create multiple mini-partition-groups; each mini-partition-group is pointed to by one or more hashtable entries.

In extendible hashing, the hashtable size (i.e., number of entries in the hashtable) is given by a parameter called the *global depth* d; the hashtable uses d least significant bits of some adopted hash function $h(k)$, and has size 2^d entries. Each mini-partition-group, that is meant to be a bucket in a hashtable, is assigned a local depth d'. The number of hashtable entries pointing to a bucket is given by $2^{d-d'}$; the d' least significant bits of these entries are the same. When the size of a mini-partition-group, with local depth less than the global depth, is larger than 2θ, we split the mini-partition-group by assigning half of the $2^{d-d'}$ entries to each new mini-group and distributing the tuples accordingly; the local depth of the newly created partitions are increased by one. To split a mini-partition-group with the local depth same as the global depth, the total entries in the hashtable should be doubled first, increasing the global depth; now that the local depth is less than the global depth, the mini-group can be splitted using the same approach as before.

When the size of a mini-partition-group is less than θ, the bucket is merged with its *buddy* bucket, if any, provided the sum of the sizes of two buckets is less than 2θ and the local depths of the two buckets are same. Let l_{bud} be the first entry of the buddy of a bucket with starting entry l; let the local and the global depth be d' d, respectively.

$$l_{bud} = \begin{cases} l + 2^{d-d'}, & \text{if } 2^{d-d'+1} \text{ divides } l \\ l - 2^{d-d'}, & \text{otherwise} \end{cases}$$

Within each mini-partition-group, the incoming tuples are joined using a simple Block-Nested Loop join. The tuples within each window partition should be expired periodically. Thus, the tuples should maintain the temporal order in the stream; this constraint makes any sort-based algorithm infeasible. At first look it appears that tuples in a window partitions can be sorted using a out-of-place storage that stores the sorted tuple IDs, whereas the actual tuples reside in the window partition. Such an approach suffers from both storage overhead and computation overhead due to frequent expiration of the stream tuples.

7.5 Scalability issues

In this section, we consider the system performance with a large number of processing nodes. Considering the scalability issue, we describe two techniques to control system overheads.

7.5.1 Degree of Declustering

Determining the appropriate degree of declustering (i.e., number of slave nodes) is an important issue while using intra-operator parallelism. Selecting low degree of declustering can lead to under-utilization of the system (i.e., nodes not actively participating in the processing waste their idle CPU cycles), and reduce system performance, overloading the processing nodes. On the other hand, high degrees of parallelism may under-utilize the processing nodes and increase communication overhead. To decrease communication overhead, the payload (i.e., number of stream tuples) of the messages sent, in each distribution epoch, to every slave node should be as high as possible [14].

Setting the upper bound of the degree of parallelism based on bounding the communication cost as in [51] is infeasible in the scenario of continuous queries for a number of reasons: firstly, unlike traditional queries, the degree of declustering may vary during execution. Secondly, the communication cost during the execution can not be estimated by a fixed, simple model. Therefore, an adaptive approach is necessary. Moreover, in a multi-process environment, measuring the execution time and communication delay is impossible due to frequent context switching by the operating system, which is transparent to the processes or threads. Based on these observations, we propose a simple approach to maintain the degree of declustering.

Our approach adjusts the degree of declustering based on the observation of the loads of the processing nodes. The master node decreases the processing nodes when the load of the processing nodes are very low, and increases the active slave nodes when the loads of the processing nodes are high. Our approach keeps the system minimally overloaded by ensuring at least S_{low} suppliers in the system. If all the nodes are either *neutral* or *consumer*, the master node decreases the degree of declustering. This minimizes the underutilization of the active slave nodes. On the other hand, the master node increases the degree of declustering when the number of suppliers in the system is greater than a fraction β of the number of consumers in the system, that is,

$$N_{sup} > \beta N_{con}$$

Here, N_{sup} and N_{con} are, respectively, the number of supplier and consumer in the system; and

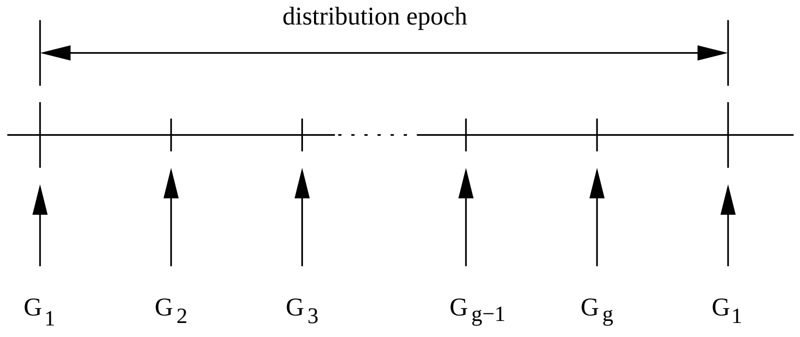


Figure 7.6: Subgroup communication

β ($0 < \beta < 1$) is a granularity parameter. This ensures the proper utilization of the system. The degree of declustering is adjusted by δ number of nodes (i.e., δ is the base correction amount).

7.5.2 Sub-group Communication

The master node distributes the buffered tuples to all the slaves at the start of every distribution epoch. As the tuples are sent to every node in a serial order, such an approach might increase average idle time while waiting for the tuples. For example, in a system with N slaves, a slave, in the worst case, should have wait in idle until the master node transmits the tuples to other $N-1$ slaves. To minimize such an overhead, we divide the slave into a number (n_g) of groups, and allow the slaves within each sub-group communicate with the master at a time; the distribution epoch is divided into n_g slots, and each sub-group receives tuples from the master within a slot assigned to the group.The details of the communication is shown in Figure 7.6.

Such a communication in sub-group also minimizes the total storage or memory required to buffer the pending tuples at the master node. Let us consider a stream S_i with a uniform rate r_i, and assume that the master distributes an equal number of tuples to every slaves. The total tuples of stream S_i arriving during a distribution epoch t_d is $t_d r_i$. With sub-group communication, the maximum buffer size for the stream at the master node, can be calculated as,

$$\begin{aligned}
M_{buf} &= \frac{r_i}{n_g}t_d + \frac{r_i}{n_g}\left(t_d - \frac{t_d}{n_g}\right) + \ldots \\
&\quad + \frac{r_i}{n_g}\left(t_d - \frac{t_d}{n_g}(n_g - 1)\right) \\
&= \frac{r_i}{n_g}\sum_{k=1}^{n_g}\left[t_d - \frac{t_d}{n_g}(k-1)\right] \\
&= \frac{r_i t_d}{n_g}\left[n_g - \frac{1}{n_g}\sum_{k=0}^{n_g-1}(k)\right] \\
&= \frac{r_i t_d}{2}\left(1 + \frac{1}{n_g}\right)
\end{aligned}$$

From this equation, it is obvious that for a large value of n_g, the maximum buffer size can be reduced almost by a half.

7.6 Experiments

This section describes the methodologies for evaluating the performance of load diffusion in executing a stream join operator, and presents experimental results demonstrating the effectiveness of the proposed load diffusion system based on synchronization of the communication among the nodes.

7.6.1 Experimental Methodology

The following paragraphs describe the major components of our experimental setup: the join techniques we consider, the data sets and the evaluation metrics and the experimental platform.

Join Processing Technique. As observed earlier in the paper, processing sliding window joins requires the maintenance of the temporal order of the tuples within a window. So, within each window partition, we apply a simple Nested Loop Join algorithm. We tune the sizes of the window partitions applying the fine tuning technique described in the paper.

Data Stream Generation. We evaluate the performance of the load diffusion algorithm, using synthetic data streams. The streams tuples are generated online during system activity. The stream tuples are generated in real time within the master node using a separate module. The

Parameter	Defaults	Comment
$W_i(i = 1, 2)$	10	Window length(min)
λ	1500	Avg. arrival rate(tuples/sec)
b	0.7	skewness in join attribute values(for *b-model*)
Th_{con}	0.01	Consumer Threshold
Th_{sup}	0.5	Supply Threshold
θ	1.5	partition tuning parameter (MB))
	4	Block Size (KB)
t_d	2	Distribution epoch (sec)
t_r	20	Reorganization epoch (sec)

Table 7.1: Default values used in experiments (Parallelizing stream joins)

stream generation modules are scheduled during idle period, within each distribution epoch, after the master has already sent the pending tuples to the slaves.

We assume that tuples within a stream S_i arrive with a Poisson arrival rate λ_i. The inter-arrival time for each tuple is given by the Poisson process. Each stream tuple has a length of 64 bytes. The domain of the join attribute A is taken as integers within the range $[0 \dots 10 \times 10^6]$. The distribution of the join attribute values for the stream tuples is captured using *b-model* [137], which is closely related to the "80/20 law" in databases [65].

Evaluation Metrics. We evaluate the performance of the system based on the capacity of the system, that is, the maximum stream rates that overload the system. We provide an indication of the capacity of the system, measuring a number of parameters: *average production delay*, communication time (or overhead), total CPU time, and the window size within a node. We measure the production delay of an output tuple as the interval elapsed since the arrival of the joining tuple with the more recent timestamp. For example, if tuples s_1 and s_2 are the joining tuples of the output tuple (s_1, s_2), where $s_1.t > s_2.t$ (s_1 being the more recent one) and current time is T_{clock}, then the delay in producing the output tuple is $(T_{clock} - s_1.t)$. This metric (i.e., average production delay) indicates how quick an output tuple is generated. Thus, this metric also provide an indication of the capacity of the system: when the system is overloaded, the incoming tuples stays a longer period of time in buffer, resulting in a larger production delay.

Experimental Platform. We have performed our experiments on cluster of machines connected by a Gigabit Ethernet Switch. Each machine has two Pentium III (coppermine) 930 MHz processors, 256 KB L2 cache, and 512 MB of main memory. For each experimental setting, we run the system for 20 minutes, and refresh the observed parameters by the elapse of a time of 10 minutes. At the master size, we provide the level of indirection, while distributing the tuples, by maintaining 60 partitions; each partitions stream or window partitions are fine

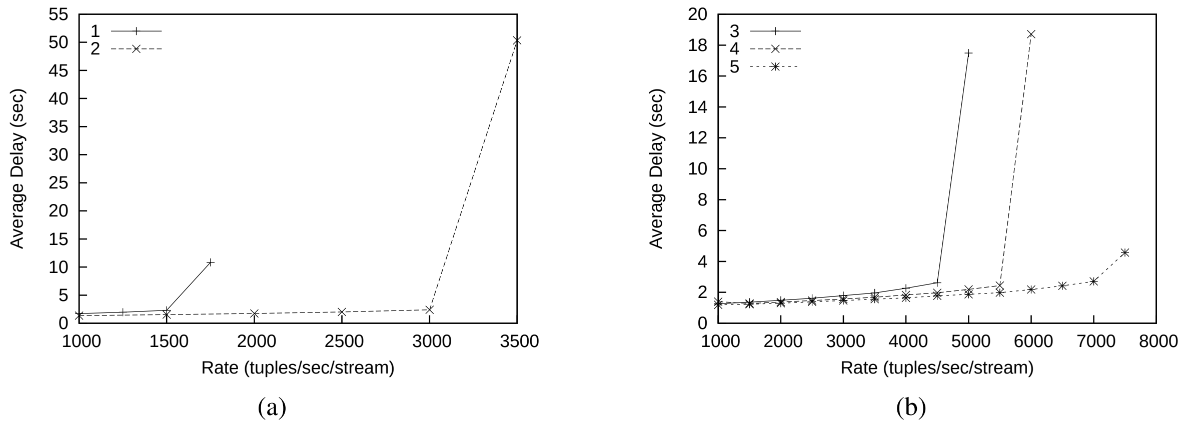


Figure 7.7: Average delay with varying stream rates (for different slave population)

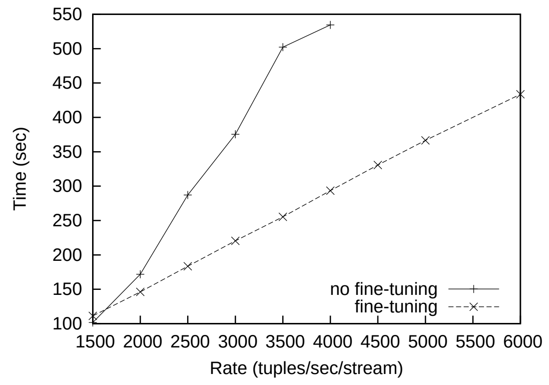
tuned, at the slave side, based on a θ-value of 1.5MB. We fix the block size to 4KB. We allocate 1MB of memory to buffer the stream tuples. We assume that each processing node has enough memory to hold the window partitions; extension of the system to cope with memory limited nodes is straightforward, based on the incorporation of the memory occupancy information during partition reorganizations. The distribution epoch and the reorganization epoch are taken as 2 and 20 seconds, respectively.

We implement the join processing algorithm in Java. We use *mpiJava* [14], a pure Java interface to MPI, that uses services of a native MPI implementation. We use LAM/MPI [23] (version 7.0.6) as the underlying native MPI implementation. Unless otherwise stated, the default values used in the experiments are as given in Table 7.1.

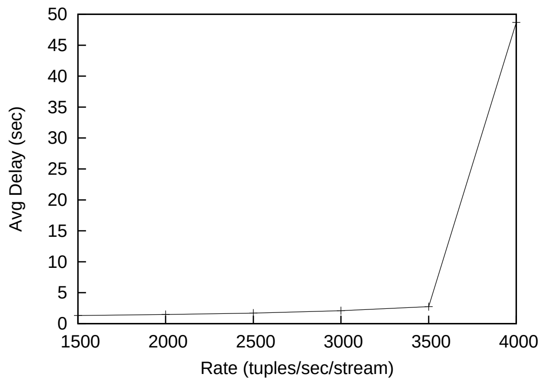
7.6.2 Experimental Results

In this section, we present a series of experimental results for assessing the effectiveness of the proposed hashjoin algorithm. We measure the average delay in generating an output tuple, maximum window sizes in the nodes, processing time, and communication overhead. For each set of experimentation, we run the system for 20 minutes. We start to gather performance data after an startup interval of 10 minutes is elapsed.

We first present the experimental results with varying stream arrival rates. Figure 7.7 shows the average delay, with varying stream arrival rates, observed in the output tuples. Each plot in the figures corresponds to different slave population. Given a number of slave nodes, the average delay increases sharply at a point where the applied load overloads the whole system.



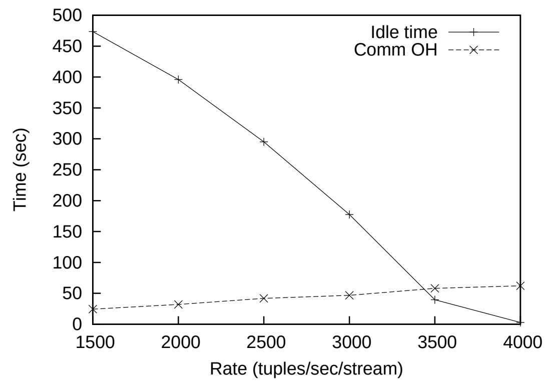
(a) Average processing time (with and without fine-tuning)



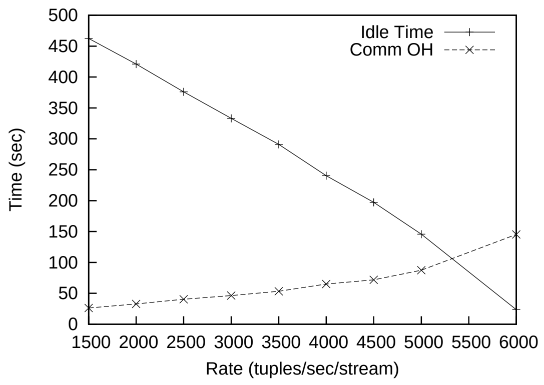
(b) Average delay (without fine-tuning)

Figure 7.8: Average processing time (i.e., CPU time) and Average delay with varying stream rates (total slave nodes=4)

Before such a saturation point, the average delay shows very little variations with the increase in stream rates. This is due to the diffusion of the loads from a temporarily overloaded node. From the two figures, we observe that the stream arrival rate which overloads the system increases as more and more slave nodes are added in the system.



(a) Without fine-grained partition tuning)



(b) with fine-grained partition tuning

Figure 7.9: Idle time and communication overhead with varying stream arrival rates (total slave nodes=4))

The effectiveness of fine grained partition tuning at the slave nodes can be observed from the Figure 7.8. Figure 7.8(a) shows the average CPU times (both with and without fine tuning at the slaves) while processing stream joins with varying arrival rates. Without fine tuning, the average CPU time required to process the joins increases sharply with the increase in the

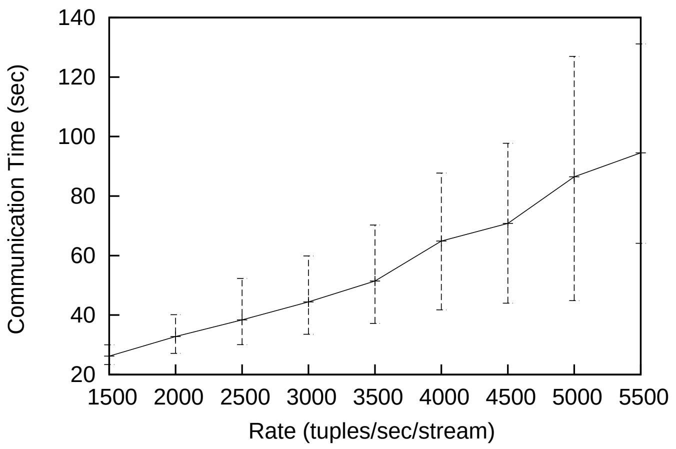


Figure 7.10: Communication overhead with varying stream rates (total slave nodes=4)

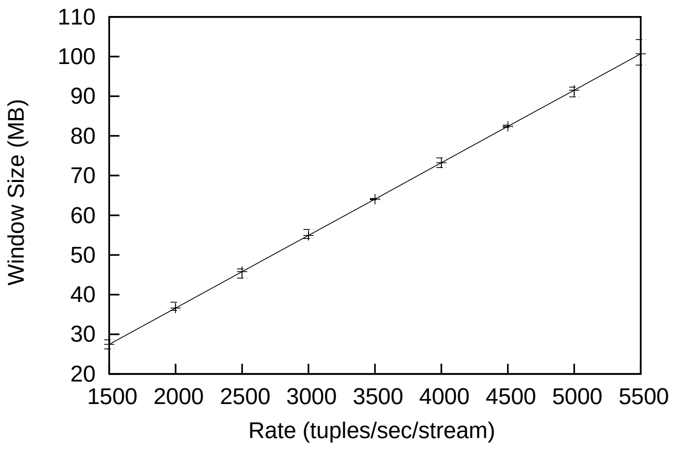


Figure 7.11: Variations in window size with varying stream arrival rates

stream rates. As shown in Figure 7.8(b), without fine tuning, the average delay is around 48 sec for a per-stream rate of 4000 tuples/sec. On the other hand, with fine tuning, the average delay for the same system (with 4 slave nodes and for an arrival rate of 4000 tuples/sec) drops to around 2 seconds (cf., Figure 7.7(b)).

Figure 7.9 shows the idle time and the communication overhead for the system with and without applying fine grained partition tuning. As shown in Figure 7.9(a), without partition tuning, the idle time for the system drops near to zero for an arrival rate of 4000 tuples/sec/stream. On the other hand, with fine grained partition tuning,in Figure 7.9(b), the idle time for the slave nodes is near to zero while the arrival rate is 6000 tuples/sec/stream. With the increase in arrival rates, the sizes of the partitions of each stream window also increases. Thus scanning the window partitions to join the incoming stream tuples consume higher CPU times

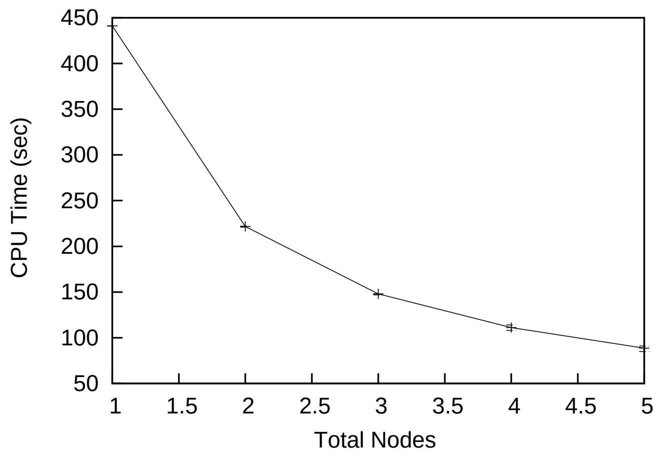


Figure 7.12: Processing times with varying nodes

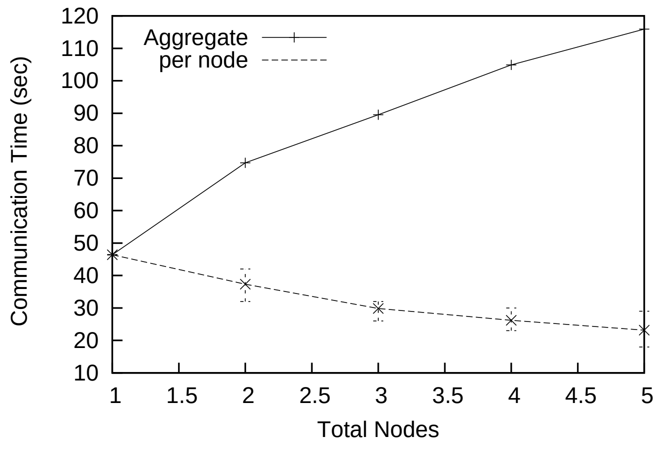


Figure 7.13: Communication overhead with varying nodes

with the increase in the stream rates. Tuning the partition sizes at the slave nodes significantly lowers the CPU time to process the join. Such a fine grained tuning incurs no communication overhead as evident from Figure 7.9(a) and Figure 7.9(b).

Figure 7.10 shows the communication overhead across the slave nodes. It shows the minimum, maximum and average communication overhead over all the slave nodes in the system. The communication time is not uniform across the slaves, as the tuples are distributed to the slaves, during a distribution epoch, in a serial order. The divergence in communication overhead across the slaves increases with an increase in arrival rates. Such divergence across the slave nodes can be minimized by maintaining a fixed order while distributing tuples across the slaves; now, the slave node can delay its connection initiation according to its position in the sequence.

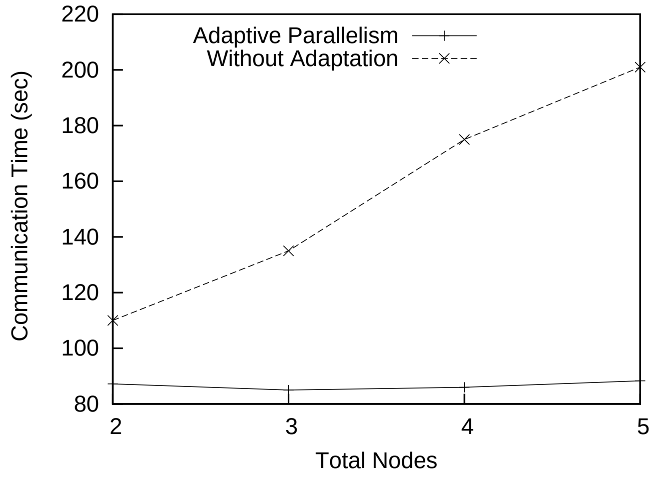
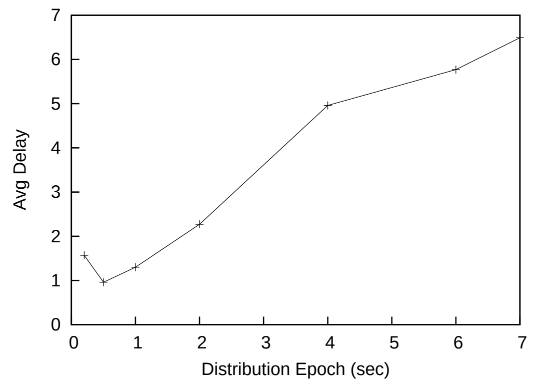


Figure 7.14: Communication overhead with varying nodes (stream arrival rate=2000 tuples/sec/stream)

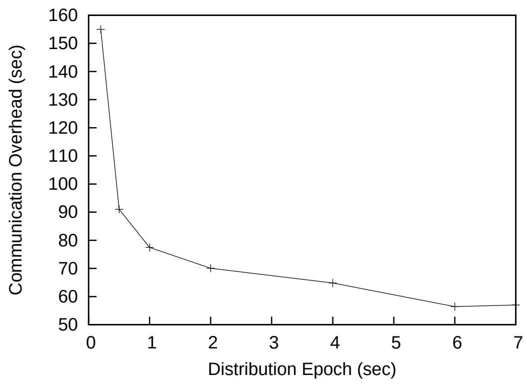
Figure 7.11 shows the sizes of the stream windows at each slave nodes. Here, the window size increases with the increase in the stream rates, and the deviations in the window sizes across the nodes is not very high. Such bounds in the deviations of the window sizes are due to the adaptive partitioning of the streams across the slave nodes.

Figure 7.12 shows the variations in the average CPU time of the slave nodes as the slave population is increased. The CPU time, as expected, decreases almost linearly. Here, the stream rate is 1500 tuples/sec/stream. However, while parallelizing the join, with low stream rates, over a large number of slaves, there might arise processing overhead due to the lack of proper amortization opportunity; for example, at a slave node, join processing might be invoked frequently at granularity less than the block size. Figure 7.13 shows the communication overhead at each node with varying slave nodes. The average communication time for a node decreases with the increase in degree of declustering; however aggregate overhead over all the slaves increases linearly. On the other hand, with adaptive parallelism (Figure 7.14), the communication overhead remains stable as the number of slave nodes is increased. This scenario makes it evident that the degree of declustering of the system should not be increased unless necessary, i.e., all the nodes should operate as close to its processing capacity as possible.

Figure 7.15 shows the average delay of the output tuples and the average communication overhead across the slave nodes with varying distribution epochs. In Figure 7.15(a), as the distribution epoch decreases, the average delay also decreases due to the decrease in the wait time at the master node. However, with the decrease in the distribution epoch, the communication overhead increases (Figure 7.15(b)) and reaches at point (not shown in the figure) where



(a) Average production delay)



(b) Communication overhead

Figure 7.15: Average production delay and communication overhead with varying distribution epochs (total slave nodes=3)

the slaves are engaged only in communication, leaving no time for the processor to process the incoming tuples. Thus, for a fixed stream arrival rate, there exists a tradeoff between the distribution epoch and the communication overhead.

7.7 Summary

In this chapter, we present a new technique to achieve fine-grained, intra-query parallelism while processing a sliding window operator over a cluster of processing nodes. The proposed algorithm balances the join processing loads over a shared-nothing cluster. Implementing the system using reliable or persistent communication poses various scalability challenges unforeseen in the previous or existing approaches; For example, maintaining all-time, any-to-any connection is infeasible in such a scenario, that requires a predefined protocol to enable the participating processes communicate at the application level. We analyze the issue of scaling the intra-query parallelism over a large number of nodes in a multi-query, multi-user environment, and propose techniques to dynamically maintain the degree of declustering, optimizing the processing and communication overheads of the system. Our experimental results demonstrate the effective of the algorithm.

Chapter 8

Conclusions and Future Works

8.1 Thesis Overview

This thesis addresses several challenges pertaining to answering exact results for join queries over data streams. Data Stream Management Systems (DSMSs) are often constrained in terms of resources such as memory and computation. To this end, contrary to load shedding, we proposed, in Chapter 3, a brute force technique to smooth loads of a stream join. The EWJ algorithm incorporates secondary storage to spill states of a join operator, refines output results by properly retrieving the disk resident data, and maximizes output rate by employing techniques to manage the memory blocks. We observed that incorporating the disk storage, the memory footprint of a stream join operator can be minimized significantly without degrading the join performance (e.g., average delay). Existing approaches based on load shedding can not provide results with an arbitrary precision. Experimental results show that, for the EWJ algorithm, the average delay and disk-I/O time is low even for an arrival rate of 250 tuple/sec/stream, when the maximum footprint of the join operator is five times the allocated memory.

In Chapter 5, we presented a join processing algorithm based on hash partitioning. The AH-EWJ algorithm presented in chapter 5 dynamically adjusts memory allocations at multiple levels: a partition level and a stream window level. Such an adaptive allocation of memory based on both the productivity and stream arrival rates minimizes disk dumps and increases the output generation rate. We proposed a generalized framework to keep highly productive blocks in memory and to maintain the blocks in memory during systems activity, forgoing any specific model of stream arrivals (e.g., age-based or frequency based model [125]). The AH-EWJ algorithm eliminates expensive disk-to-disk phase, and amortizes a disk scan over a large number of input tuples, rendering the algorithm I/O-efficient. Algorithms for processing joins over finite streams (i.e., XJoin [133] or its variants) can not be applied directly to process windowed stream joins. Also, the AH-EWJ algorithm presents a technique to eliminate random partition thrashing by properly marshalling the blocks, by bounding the divergence of the block numbers, while dumping onto the disk. In chapter 4 we explained the necessary changes to RPJ [129], which is the most efficient one among the variants of the XJoin. We used the RPWJ algorithm as a baseline one, and compared the performance of the AH-EWJ algorithm, proposed in Chapter 5, with that of the RPWJ. The experimental results demonstrate the effectiveness of the proposed algorithm in a resource limited environment. For example, incorporating a disk storage, the memory footprint of a stream join operator can be minimized by an order of magnitude(~90%). In a system with numerous continuous queries, such a memory-limited algorithm increases the scalability of the system.

In Chapter 6, we considered a hybrid join operator that involves both a data stream and a persistent relation. We proposed a partition-based join algorithm to compute a hybrid join between a fast, time varying or bursty update stream and a persistent relation using limited memory. Such a join operator is the crux of a number of common transformations (e.g., surrogate key assignment, duplicate detection etc.) in an active data warehousing. The partition-based approach, proposed in Chapter 6, minimizes the processing overhead, exploits the spatio-temporal locality within the update stream, minimizes delays in output tuples, and amortizes a disk access over a large number of stream tuples. The experimental results show a significant performance gain: The proposed algorithm reduces the response time by an order of magnitude; also, the algorithm reduces both the processing overhead and disk-I/O time.

In Chapter 7, we turned to the setting of a distributed system, where a number of non-dedicated, and possibly heterogeneous, processing nodes are connected through a high speed network. We proposed a new technique to achieve fine-grained, intra-query parallelism while processing a stream join operator over a shared nothing cluster. We proposed a framework, based on a fixed or predefined sequence of communication, to distribute the join processing loads over the shared nothing cluster. We analyzed the issue of scaling the intra-operator parallelism over a large number of processing nodes in a multi-query, multi-user environment, and proposed techniques to dynamically maintain the degree of declustering, optimizing the processing and communication overheads of the system. We propose experimental results that show the effectiveness of the adaptive algorithm in reducing both the communication and processing overheads while balancing loads in a Shared Nothing Cluster.

8.2 Future Directions

Looking beyond the specific contributions of this thesis, there exists numerous issues that are worthy of further investigation. In this section, we provide a few directions for future works.

8.2.1 Disk-based continuous queries

In traditional DBMSs, streams of queries are processed over non-streaming database. On the other hand, In data stream management system, ad-hoc queries are continuously added and removed from the system. Thus, both the queries and data in a data stream management system are streams in natures as they arrive continuously in time. Considering the streaming characteristics of both the queries and data, PSoup [30] proposes a novel query query engine that combines the processing of both ad-hoc and continuous queries, and treats both the queries and data analogously. The query engine maintains *State Modules* (SteMs) for both data and queries, and executes the streams of queries by transforming the query plan into a join operator between the query SteM and data SteM. However, PSoup implements such a join operation in main memory and doesn't consider disk storage. Incorporating disk-based storage raises the of swapping both data and queries, from the data and query SteM respectively, between memory and the disk. In such a system, some queries are executed more frequently than others. Based on the frequency of execution of the queries, the scheduling mechanism should de-schedule both and queries out of the memory. Such an issue of processing stream of queries over data streams incorporating disk storage is a significant research problem.

8.2.2 Band-Join Processing

The join algorithm AH-EWJ proposed in Chapter 5, applies a hash function to partition the stream. Such a hash-based technique can not be used while processing band joins [2], which requires a range partitioning over the domain of the join attributes. Devising a framework for processing band joins is a research issue that worth further investigation.

8.2.3 Concurrent hybrid queries

In an active data warehouse multiple queries execute concurrently, which consumes the disk relations. Thus, there lies the opportunity to share a disk scan of an input table over all the concurrent queries processing the same table. Such a sharing of disk scans requires a new processing mechanism for maintaining the allocated memories. Such a mechanism has the benefit of sharing the disk scan cost, the join processing cost, and the available memory. Such shared processing increases the query throughput and minimizes the system loads.

8.2.4 Distributed Streams

This dissertation does not consider the issue of processing joins over distributed streams. In this scenario, the stream sources are located at different places within the network, and the join algorithm should process join over the streams distributed in the network. The streams from different sources need not be sent to a central server, or all the streams tuples from every sources should not be sent to all other sources. Reference [86] proposes such an approximate join processing algorithm based on discrete Fourier transforms (DFTs). This algorithm adjusts the number of tuples sent between two sources, over the network, based on correlation of streams from the respective sources. Each nodes takes a probabilistic approach, based on the DFT co-efficients received from different sources, in determining how to distribute tuples to the remaining nodes. In such a setting, processing accurate results of the joins over the distributed streams is a significant research problem. We speculate that both the network and processing cost can be minimized using bitmap indices [28] instead of DFTs. Such a join processing technique requires a bitmap encoding scheme that should adjust the bitmap indices based on the arrival pattern of tuples within different streams.

8.2.5 Parameter Tuning in Load diffusion

The load diffusion algorithm proposed in Chapter 7 parallelizes a join operator over multiple nodes based a few parameters, e.g., distribution epoch, subgroup size, etc. While deployed over a large number of processing nodes, the network and processing overheads depend on the subgroup size and the distribution epoch. Dynamically tuning these parameters is a challenging research problem. Moreover, we did not consider disk-I/O at the local nodes. Incorporating disk-I/Os at the local nodes is another topic for future work.

References

[1] Daniel J. Abadi, Samuel Madden, and Wolfgang Lindner. REED: Robust, efficient filtering and event detection in sensor networks. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 769–780, 2005.

[2] Pankaj K. Agarwal, Junyi Xie, Jun Yang, and Hai Yu. Monitoring continuous band-join queries over dynamic data. In *Proc. Intl. Symposeum on Algorithms and Computation (ISAAC)*, pages 349–359, 2005.

[3] Yanif Ahmad, Ugur Çetintemel, John Jannotti, and Alexander Zgolinski. Locality aware networked join evaluation. In *Intl. Workshop on Networking Meets Database (NetDB*, page 1183, 2005.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[5] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, Dallas, Texas, USA, May 2000.

[6] Ahmed Ayad, Jeffrey Naughton, Stephen Wright, and Utkarsh Srivastava. *Approximating Streaming Window Joins Under CPU Limitations*. Computer Science Department, Technical Report #1542, University of Wisconsin, Medison, November 2005.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 253–264, 2003.

[8] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Processing sliding window multi-joins in continuous queries over data streams. In

Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), pages 1–16, Madison, Wisconsin, USA, June 2002.

[9] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, pages 238–249, 2005.

[10] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, Paris, France, June 2004.

[11] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *Proc. Intl. Conf. on Data Engineering*, pages 118–129, Tokyo, Japan, April 2005.

[12] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(1):109–120, March 2001.

[13] Shivnath Babu and Jennifer Widom. Streamon: An adaptive engine for stream query processing. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 931–932, Paris, France, June 2004.

[14] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpiJava: An object-oriented java interface to MPI. In *Intl. Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*, San Juan, Puerto Rico, April 1999.

[15] M. Balazinska, H. Balakrisnan, and M. Stonebraker. Content-based load management in federated distributed systems. In *Proc. Symposium on Networked System Design and Implementation (NSDI)*, March 2004.

[16] M. Balazinska, A. Deshpande, M. J. Franklin, P. B. Gibbons, J. Gray, S. Nath, M. Hansen, M. Liebhold, A. Szalay, and V. Tao. Data management in the worldwide sensor web. *IEEE Pervasive Computing*, 6(2):30–40, 2007.

[17] Mostafa Bamha and Gaétan Hains. Frequency-adaptive join for shared nothing machines. *Parallel and Distributed Computing Practices*, 2, 1999.

[18] G. Barrenetxea, F. Ingelrest, G. Schaefer, M. Vetterli, O. Couach, and M. Parlange. Sensorscope: Out-of-the-box environmental monitoring. In *Proc. Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, pages 332–343, St. Louis, Missouri, USA, April 2008.

[19] M. W. Blasgen and K. P. Eswaran. Storage and access in relational databases. *IBM Systems Journal*, 16(4):362–377, 1977.

[20] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world. *IEEE Personal Communications*, 7:10–15, 2000.

[21] Mihaela A. Bornea, Vasilis Vassalos, Yannis Kotidis, and Antonios Deligiannakis. Double index nested-loop reactive join for result rate optimization. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 481–492, 2009.

[22] D. Burleson. New developments in oracle data warehousing. *Burleson Consulting*, April 2004.

[23] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[24] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 838–849, Berlin, Germany, 2003.

[25] Donald Carney, Ugur ÃĞetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams – a new class of data management applications. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 215–226, Hong Kong, China, August 2002.

[26] Ugur Cetintemel and William Hobbib. Getting real with forex. Futures Magazine, May 2007.

[27] Abhirup Chakraborty and Ajit Singh. A partition-based approach to support streaming updates over persistent data in an active data warehouse. In *To Appear Proc. IEEE Intl. Symp. on Parallel and Distributed Processing (IPDPS)*, pages 1–11, Rome, Italy, May 2009.

[28] Chee Yong Chan and Yannis E. Ioannidis. Bitmap index design and evaluation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 355–366, 1998.

[29] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[30] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 203–214, Hong Kong, China, August 2002.

[31] Sirish Chandrasekharan. *Query Processing over Live and Archived Data Streams*. PhD Thesis, UC Berkeley, 2005.

[32] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, Dallas, Texas, May 2000.

[33] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2003.

[34] V. Chvatal. A greedy-heuristic for set covering problem. *Mathematics of Operations Research*, 4(3):233–235, August 1979.

[35] John Cieslewicz, Kenneth A. Roos, and Ioannis Giannakakis. Parallel buffers for chip multiprocessors. In *Proc. Intl. Workshop on Data Management on New Hardware (DaMoN)*, pages 9–18, June 2007.

[36] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2003.

[37] Owen Cooper, Anil Edakkunni, Michael J. Franklin, Wei Hong, Shawn R. Jeffery, Sailesh Krishnamurthy, Frederick Reiss, Shariq Rizvi, and Eugene Wu 0002. Hifi: A unified architecture for high fan-in systems. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 1357–1360, 2004.

[38] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, and Frederick Smith. Hancock: A language for analyzing transactional data streams. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 26(2):301–338, 2004.

[39] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 647–651, San Diego, USA, 2003.

[40] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 40–51, San Diego, USA, June 2003.

[41] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 27–40, 1992.

[42] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 299–310, Hong kong, China, August 2002.

[43] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. On producing join results early. In *Proc. ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 134–142, 2003.

[44] Nick G. Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Ntworking*, 9(3):280–292, 2001.

[45] Jakob Eriksson, Lewis Girod, Bret Hull, Ryan Newton, Samuel Madden, and Hari Balakrishnan. The pothole patrol: using a mobile sensor network for road surface monitoring. In *Proc. Intl. Conf. on Mobile systems, applications, and services*, pages 29–39, Breckenridge, CO, USA, 2008.

[46] Arvind Arasu et al. Linear road: A stream data management benchmark. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 480–491, Toronto, Canada, September 2004.

[47] Daniel J. Abadi et al. The design of the borealis stream processing engine. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[48] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, Septemebr 1979.

[49] Anton Faradjian, Johannes Gehrke, and Philippe Bonnet. Gadt: A probability space adt for representing and querying the physical world. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 201–211, 2002.

[50] Caceres Duffield Feldmann and R. Caceres et al. Measurement and analysis of ip network usage and behavior. *IEEE Communications Magazine*, 38:144–151, 2000.

[51] Minos N. Garofalakis and Yannis E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 365–376, 1996.

[52] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Adaptive load shedding for windowed stream joins. In *Proc. Intl. Conf. on information and Knowledge Management (CIKM)*, pages 171–178, Bremen, Germany, November 2005.

[53] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. A load shedding framework and optimizations for m-way windowed stream joins. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 536–545, Istanbul, Turkey, April 2007.

[54] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1):46–55, 2004.

[55] Thanna M. Ghanem. Exploiting predicated-window semantics over data streams. *SIGMOD Record*, 35(1):3–8, March 2006.

[56] Garth A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):3–8, November 2000.

[57] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 79–88, Roma, Italy, 2001.

[58] Lukasz Golab, Shaveen Garg, and M. Tamer Ozsu. On indexing sliding windows over online data streams. In *International Conference on Extending Database Technology*, pages 712–729, Heraklion, Crete, Greece, March 2004.

[59] Lukasz Golab, Theodore Johnson, and Vladislav Shkapenyuk. Scheduling updates in a real-time stream warehouse. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 1207–1210, 2009.

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

[61] Lukasz Golab and Tamer Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 500–511, Berlin, Germany, September 2003.

[62] Hector Gonzalez, Jiawei Han, Xiaolei Li, and Diego Klabjan. Warehousing and analyzing massive rfid data sets. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, page 83, 2006.

[63] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, June 1993.

[64] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. Intl. Conf. on Management of Data*, pages 102–111, Atlantic City, NJ, USA, May 1990.

[65] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 243–252, May 1994.

[66] Xiaohui Gu, Z. Wen, C. Y. Lin, and P. S. Yu. An adaptive distributed video correlation system. In *Proc. ACM Multimedia*, 2006.

[67] Xiaohui Gu, Philip S. YU, and Hiaxun Wang. Adaptive load diffusion for multiway windowed stream joins. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 146–155, Istanbul, Turkey, April 2007.

[68] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[69] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 287–298, Philadelphia, Pennsylvania, USA, June 1999.

[70] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proc. 15th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 75–84, 2003.

[71] Lilian Harada and Masaru Kitsuregawa. Dynamic join product skew handling for hash-joins in shared nothing database systems. In *Proc. Intl. Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 246–255, April 1995.

[72] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182, Tucson, Arizona, USA, May 1997.

[73] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, USA, 2000.

[74] Kien A. Hua and Chiang Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 525–535, Bercelona, Catalona, Spain, 1991.

[75] K. Imasaki, H. Nguyen, and S. P. Dandamudi. An adaptive hash join algorithm on a network of workstations. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[76] Milena Ivanova and Tore Risch:. Customizable parallel execution of scientific stream queries. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 157–168, Trondheim, Norway, August 2005.

[77] Zachary G. Ives, Amol Deshpande, and Vijayshankar Raman. Adaptive query processing: Why, how, when, and what next? In *Proc. Intl. Conf. on Very Large Databases (VLDB*, pages 1426–1427, Vienna, Austria, September 2007.

[78] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proc. Intl. Conf. on Mobile Computing and Networking*, pages 271–278, Seattle, Washington, USA, 1999.

[79] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *Proc. Intl. Conf. on Data Engineering*, pages 341–352, Bangalore, India, March 2003.

[80] Alexandros Karakasidis, Panos Vassiliadis, and Evaggelia Pitoura. ETL queues for active data warehousing. In *Proc. Intl. Workshop on Information Quality in Informational Systems (IQIS)*, pages 28–39, 2005.

[81] Arthur M. Keller and Shaibal Roy. Adaptive parallel hash join in main-memory databases. In *Proc. Intl. Conf. on Parallel and Distributed Information Systems*, pages 58–67, 1991.

[82] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steve Glaser, and Martin Turon. Wireless sensor networks for structural health monitoring. In *Proc. Intl. Conf. on on Embedded Networked Nensor Systems (SenSys)*, pages 427–428, Baltimore, MD, USA, 2006.

[83] Masaru Kitsuregawa and Yasushi Ogawa. Bucket spreading parallel hash: a new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 210–221, Brisbane, Australia, 1990.

[84] Jon M. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 91–101, Edmonton, Alberta, Canada, 2002.

[85] Flip Korn, S. Muthukrishnan, and Yihua Wu. Modeling skew in data streams. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 181–192, Chicago, Illinois, USA, 2006.

[86] Vassil Kriakov, Alex Delis, and George Kollios. Approximate data stream joins in distributed systems. In *Proc. Intl. Conf. on Distributed computing Ststems (ICDCS)*, page 5, 2007.

[87] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 1996.

[88] W. Labio, L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient resumption of interrupted warehouse loads. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 2000.

[89] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 2000.

[90] Ramon Lawrence. Early hash join: A configurable algorithm for the efficient and early production of join results. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 841–852, 2005.

[91] Rubao Lee and Zhiwei Xu. Exploiting stream request locality to improve query throughput of a data integration system. *IEEE Trans. on Computers*, 58(10):1356–1368, October 2009.

[92] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. on Networking*, 2(1):1–15, 1994.

[93] Alberto Lerner and Dennis Shasha. The Virtues and Challenges of Ad Hoc + Streams Querying in Finance. *IEEE Data Engineering Bulletin*, 26(1):49–56, 2003.

[94] Justin Levandoski, Mohamed E. Khalefa, and Mohamed F. Mokbel. Permjoin: An efficient algorithm for producing early results in multi-join query plans. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 1433–1435, Cancun, Mexico, 2008.

[95] Feifei Li, Ching Chang, George Kollios, and Azer Bestavros. Characterizing and exploiting reference locality in data stream applications. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, page 81, 2006.

[96] Bin Liu, Yali Zhu, and Elke A. Rundensteiner. Run-time operator state spilling for memory intensive long-running queries. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 347–358, Chicago, Illinois, USA, June 2006.

[97] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[98] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 555–566, 2002.

[99] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, 2002.

[100] A. Mainwaring, J. Polastre, R. Szewcyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proc. ACM Intl. Workshop on Wireless Sensor Networks and Applications (WSNA)*, pages 88–97, Atlanta, Georgia, USA, september 2002.

[101] P. Mishra and M. H. Eich. Join processing in relational database systems. *ACM Computing Surveys*, 24(1):63–113, 1992.

[102] M Mokbel, M. Liu, and W. Aref. Hash-merge-join: A non-blocking join algorithm for producing fast and early join results. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 251–263, 2004.

[103] Rajeev Motwani and Dilys Thomas. Caching queues in memory buffers. In *Proc. Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 541–549, New Orleans, Louisiana, USA, January 2004.

[104] Rajeev Motwani, Jennifer Widom, and et al. Query processing, approximation and resource management in a data stream management system. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[105] D. Musiani, K. Lin, and T. Simunic Rosing. Active sensing platform for wireless structural health monitoring. In *Proc. Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, pages 390–399, Cambridge, Massachusetts, USA, 2007.

[106] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[107] Suman Nath, Amol Deshpande, Yan Ke, Phillip B. Gibbons, Brad Karp, and Srinivasan Seshan. Irisnet: An architecture for internet-scale sensing services. In *Proc. Intl. Conf. on Very Large Databases(VLDB)*, pages 1137–1140, Berlin, Germany, September 2003.

[108] Suman Nath, Jie Liu, and Feng Zhao. Challenges in building a portal for sensors worldwide. In *Proc. Intl. Workshop on World-Sensor-Web*, Boulder, Colorado, USA, 2006.

[109] J. Nievergelt and H. Hinterberger. The grid file: An adaptable, symmetric multiple key file structure. *ACM transactions on Database Systems*, 9(1):38–71, March 1984.

[110] Adegoke Ojewole, Qiang Zhu, and Wen chi Hou. Window join approximation over data streams with important semantics. In *Proc. Intl. Conf. on information and Knowledge Management (CIKM)*, pages 112–121, Virginia, USA, November 2006.

[111] Douglas Stott Parker, Richard R. Muntz, and H. Lewis Chau. The tangram stream query processing system. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 556–563, 1989.

[112] Douglas Stott Parker, Eric Simon, and Patrick Valduriez. Svp: A model capturing sets, lists, streams, and parallelism. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 115–126, 1992.

[113] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. Intl. Conf. on Data Engineering*, Atlanta, USA, April 2006.

[114] Thomas Plagemann, Vera Goebel, Andrea Bergamini, Giacomo Tolu, Guillaume Urvoy-Keller, and Ernst W. Biersack. Using data stream management systems for traffic analysis – a case study. In *Proc. Intl. Workshop on Passive and Active Network Measurement (PAM)*, pages 215–226, 2004.

[115] Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils-Erik Frantzell. Supporting streaming updates in an active data warehouse. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 476–485, Istanbul, Turkey, April 2007.

[116] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[117] Erhard Rahm and Robert Marek. Dynamic multi-resource load-balancng in parallel database systems. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 395–406, Zurich, Switzerland, 1995.

[118] Vijayshankar Raman, Wei Han, and Inderpal Narang. Parallel querying with non-dedicated computers. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 61–72, Trondheim, Norway, 2005.

[119] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online dynamic reordering for interactive data processing. In *Proc. Intl. Conf. on Very Large Databases (VLDB*, pages 709–720, Edinburgh, Scotland, UK, 1999.

[120] D. A. Schneider and D. J. DeWitt. A performance evaluation of four paralle join algorithms in a shared-nothing multiprocessor environment. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1989.

[121] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, 1979.

[122] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. Intl. Conf. on Data Engineering*, pages 25–36, Bangalore, India, March 2003.

[123] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transaction on Database Systems (TODS)*, 11(3):239–264, 1986.

[124] Gyula Simon, Akos Ledeczi, and Miklos Maroti. Sensor network-based countersniper system. In *Proc. Intl. Conf. on on Embedded Networked Nensor Systems (SenSys)*, pages 1–12, Baltimore, MD, USA, November 2004.

[125] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 324–335, Toronto, Canada, September 2004.

[126] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, ISBN 0-201-63346-9, 1994.

[127] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, page 594, 1996.

[128] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *USENIX Annual Technical Conference*, 1998.

[129] Yufei Tao, Man Lung Yiu, Dimitris Papadias, Marios Hadjieleftheriou, and Nikos Mamoulis. RPJ: Producing fast join results on streams through rate-based optimization. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 371–382, Baltimore, Maryland, USA, June 2005.

[130] Nesime Tatbul, Ugur ÃĞetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 309–320, Berlin, Germany, September 2003.

[131] Stanford STREAM Team. *SQR—A Stream Query Repository*, 2002. http://infolab.stanford.edu/stream/sqr/.

[132] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. Continuous queries over append-only databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, San Diego, California, USA, June 1992.

[133] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[134] Ymir Vigfusson, Adam Silberstein, Brian F. Cooper, and Rodrigo Fonseca. Adaptively parallelizing distributed range queries. *Proc. VLDB Endowment*, 2(1):682–693, 2009.

[135] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 285–296, Berlin, Germany, September 2003.

[136] M. Wang, S. Papadimitriou, T. Madhyastha, C. Faloutsos, and N. H. Change. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *Proc. Intl. Conf. on Data Engineering*, pages 507–516, February 2002.

[137] Mengzhi Wang, Anastassia Ailamaki, and Christos Faloutsos. Capturing the spatio-temporal behavior of real traffic data. In *IFIP Intl. Symp. on Computer Performance Modeling, Measurement and Evaluation*, Rome, Italy, September 2002.

[138] Xiaodan Wang, Randal C. Burns, Andreas Terzis, and Amol Deshpande. Network-aware join processing in global-scale database federations. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 586–595, 2008.

[139] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer Magazine*, 34(1):44–51, 2001.

[140] C. White. Intelligent business strategies: Real-time data warehousing heats up. *DM Review*, 2002.

[141] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. Intl. Conf. on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, Miami, Florida, USA, December 1991.

[142] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. A parallel hash join algorithm for managing data skew. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1355–1371, November 1993.

[143] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High performance complex event processing over streams. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, Chicago, Illinois, USA, June 2006.

[144] Junyi Xie, Jun Yang, and Yuguo Chen. On joining and caching stochastic streams. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 359–370, Baltimore, Maryland, USA, June 2005.

[145] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. Intl. Conf. on Data Engineering*, pages 791–802, Tokyo, Japan, April 2005.

[146] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and deborah Estrin. A wireless sensor network for structural monitoring. In *Proc. Intl. Conf. on on Embedded Networked Nensor Systems (SenSys)*, pages 13–24, Baltimore, MD, USA, 2004.

[147] Yu Xu, Pekka Kostama, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 1043–1052, Vancouver, Canada, June 2008.

[148] X. Zhang and E.A. Rundensteiner. Integrating the maintenance and synchronization of data warehouses using a cooperative framework. *Information Systems*, 27(4), 2002.

[149] Xiaofang Zhou, , and Maria E. Orlowska. Handling data skew in parallel hash join computation using two-phase scheduling. In *Proc. Intl. Conf. on Algorithm and Architecture for Parallel Processing*, pages 527–536, 1995.

[150] Yunyue Zhu and Dennis Shasha. Efficient elastic burst detection in data streams. In *Proc. Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 336–345, Washington, DC, USA, August 2003.

[151] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 1995.