Department of Computer Science
The University of Auckland
New Zealand

Efficient Joins to Process Stream Data

Muhammad Asif Naeem

March 2012

A thesis submitted in partial fulfillment of the requirements of Doctor of Philosophy in Science

Abstract

Data integration used to be offline, but real-time data integration has become more and more important. Research into stream databases can be naturally applied to near-real-time data integration. Several important problems in near-real-time data integration can be naturally expressed as joins. Many stream joins assume all join inputs to be streams. Recently, interest has been growing in joins with heterogeneous input, in particular joins between streams and disk-based input. MESHJOIN is a well known algorithm published in this area. The algorithm was designed particularly for application scenarios where memory resources are limited. However, the algorithm suffers from some limitations. Briefly, the memory distribution among the join components and the strategy used for accessing the disk-based data are suboptimal.

This thesis provides an independent analysis of the MESHJOIN algorithm. The focus of analysis is on equijoins as one of the most important special cases of joins. It has been shown that if a realistic distribution is assumed on stream data, such as a Zipfian distribution, MESHJOIN performs suboptimally. A set of algorithms have been developed that address the problems in MESHJOIN and they perform better than MESHJOIN in defined settings. In the end, three robust algorithms have been developed for both sorted and unsorted disk-based data. For these algorithms cost models have been developed for tuning the algorithms and validation of our implementation. An experimental study has been carried out for comparing these algorithms empirically. For that purpose a synthetic workload generator has been designed and developed. With the synthetic datasets, measurements have been taken in experiments that validate the cost models of the algorithms. The implemented algorithms are made available publicly as open source for independent analysis.

In the future this research can be extended in two directions. One is to improve the join operators further. The other is to apply the join operators in emerging application scenarios.

Acknowledgements

Firstly I am extremely thankful to Allah Almighty for giving me the strength and fortitude to accomplish this milestone.

I owe my deepest gratitude to my supervisors Prof. Gillian Dobbie and Dr. Gerald Weber. Their technical guidance, encouragement and moral support from the preliminary to the concluding level enabled me to enhance my subject knowledge and polish my research skills. Their valuable supervision will enable me to undertake challenging research problems in the future.

I would like to thank Higher Education Commission (HEC), Pakistan and The University of Auckland for giving me the opportunity to do a PhD at this premier institution and to HEC in particular for funding my research. In Computer Science Department, I am very thankful to Heather Armstrong, Sithra Sukumaar, Robyn Young and Cynthia Qu for their administrative support during my study in the department.

I also acknowledge Dr. Allison Heard from the Mathematics Department and Lisa Chen from the Statistics Department for their help in deriving the cost calculation for one of my algorithms.

This acknowledgment would not be complete without mentioning my colleagues from the Knowledge Management Group and the HEC scholar community. Their companionship has been a source of great relief and entertainment in this intellectually challenging journey.

Last but not the least I would not have been standing at the finish line had it not been for the selfless love and prayers of my parents and wife. Their affection and encouragement helped me pass through the thick and thin.

I dedicate this thesis to my little doll, Sheza Naeem.

List of Publications

The dissertation is based on the following research publications:

1. M. Asif Naeem, Gillian Dobbie, Gerald Weber, **An Event-Based Near Real-Time Data Integration Architecture**, *EDOCW'08: Proceedings of the 12th Enterprise Distributed Object Computing Conference Workshops*, IEEE Computer Society, Washington, DC, USA, 2008.

2. M. Asif Naeem, Gillian Dobbie, Gerald Weber, **Comparing Global Optimization and Default Settings of Stream-based Joins**, *BIRTE'09: VLDB Workshop*, Lyon, France, 2009.

3. M. Asif Naeem, Gillian Dobbie, Gerald Weber, Shafiq Alam, **R-MESHJOIN for Near-real-time Data Warehousing** *DOLAP'10: Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP*, ACM, Toronto, Canada, 2010.

4. M. Asif Naeem, Gillian Dobbie, Gerald Weber, **HYBRIDJOIN for Near-Real-Time Data Warehousing**, *IJDWM'11: International Journal of Data Warehousing and Mining*, IGI Global, 2011.

5. M. Asif Naeem, Gillian Dobbie, Gerald Weber, **X-HYBRIDJOIN for Near-Real-Time Data Warehousing**, *BNCOD'11: 28th British National Conference on Databases*, Manchester, UK, 2011.

6. M. Asif Naeem, Gillian Dobbie, Gerald Weber, **Optimized X-HYBRIDJOIN for Near-Real-Time Data Warehousing**, *ADC'12: Proceedings of the 23rd Australasian Database Conference*, Melbourne, Australia , 2012.

7. M. Asif Naeem, Gillian Dobbie, Gerald Weber, Imran Sarwar Bajwa, **Efficient Usage of Memory Resources in Near-Real-Time Data Warehousing**, *IMTIC'12: Proceedings of the International Multi-topic Conference*, Pakistan , 2012.

8. M. Asif Naeem, Gillian Dobbie, Gerald Weber, Imran Sarwar Bajwa, **A Parametric Analysis of Stream based Joins**, *IMTIC'12: Proceedings of the International Multi-topic Conference*, Pakistan , 2012.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Introduction

A data stream is a continuous sequence of items produced in real-time fashion. We assume these items are data records with attributes. A stream can be considered to be a relational table of infinite size. Due to this it is impossible to maintain an order of the items in the stream with respect to an arbitrary attribute. Likewise it is impossible to store the entire stream in memory. Because of these characteristics online stream processing has become a novel field in the area of data management. A number of common examples where online stream processing is important are network traffic monitoring [9, 31, 44, 76, 97], sensor data [16], web log analysis [30, 45], online auctions [7], inventory and supply-chain analysis [40, 53, 110] and real-time data integration [90, 91].

1.2 Data Stream Processing

Conventional Database Management Systems (DBMSs) are designed using the concept of persistent and interrelated data sets. These DBMSs are stored in reliable repositories, which are updated and queried frequently. But there are some modern application domains where data is generated in the form of a stream and Data Stream Management Systems (DSMSs) are required to process the stream data continuously. A variety of stream processing engines has been published in the literature [1, 6, 23, 43, 54].

The basic difference between a traditional DBMS and a DSMS is the nature of query execution. In DBMSs data is stored on disk and queries are performed over persistent data [14, 15, 29, 55, 95]. While in the stream, data items arrive online and stay in the memory for short intervals of time. DSMSs need to work in non-blocking mode while executing a sequence of operations over the data stream [6, 10, 11, 12, 17, 25, 37, 51, 56, 58, 102]. The eight important requirements for processing real-time stream data are described by Michael et al. [96]. To accommodate the execution of a sequence of operations DSMSs use the concept of a window. A window is basically a snapshot taken at a certain point in time and it contains a finite set of data items. When there are multiple operators each operator executes and stores its output in a buffer, which is further used as an input for some other operator. Therefore each operator needs to manage the contents of the buffer before it is overwritten.

Common operations performed by most DSMSs are filtering, aggregation, enrichment, and information processing. A stream-based join is required to perform these operations.

1.3 Stream-based Joins

A stream-based join is an operation to combine the information coming from multiple data sources. These sources may be in the form of streams or disk-based. Stream-based joins are important components in modern system architectures, where just-in-time delivery of data is expected. There is a number of examples where these joins play an important role. For example, in the field of networking two streams of data packets can be joined using their packet *id*'s to synchronise the flow of packets through routers [97]. Another example is an online auction system which generates two streams, one stream for opening

an auction, while the other stream consisting of bids on that auction [101, 112]. A stream-based join is required to relate the bids with the corresponding opened auction.

This thesis considers a particular class of stream-based joins, namely a join of a single stream with a slowly-changing table. Updating of indices is not included in the scope of the thesis however, there are approaches available to deal with it [26]. Such a join can be applied in real-time data warehousing [5, 20, 49, 62, 82, 88, 90, 91]. In this application, the slowly-changing table is typically a master data table. Incoming real-time sales data may comprise the stream. The stream-based join can be used, for example, to enrich the stream data with master data. The most natural type of join in this scenario would be an equijoin, performed, for example, on a foreign key in the stream data.

1.4 Motivation

The demand for real-time processing of applications with huge volumes of stream data is increasing daily. Common examples of these stream-based applications are daily supermarket transactions, network traffic monitoring, web log analysis, fraud detection etc. The processing of these stream-based applications exploits the concepts of stream-based join operators.

The large capacity of current main memories can be utilized for executing stream-based operations, as well as the considerable computing resources. For master data of the right size, for example, main-memory algorithms can be used. However, there are several scenarios where master data is huge in volume and stream joins that use minimum resources are preferable. The focus here is to deal with these kinds of scenarios. One particular scenario of interest is stream processing in mobile networks. In mobile networks, multiple operations such as correlation, aggregation, and encoding or decoding execute on a near-real-time basis. Some operations also use the information stored on disk. A stream-based join operator is required to fulfill each operation. If multiple operations are co-located on the same server while each operation needs information from the disk then, due to limited available memory for each operation, it is difficult to keep the entire disk data in memory permanently. Using join operators which are less resource-intensive would be preferable in handling these situations. Figure 1.1 presents a graphical interpretation of the scenario where end user's transactions need to be processed online with master

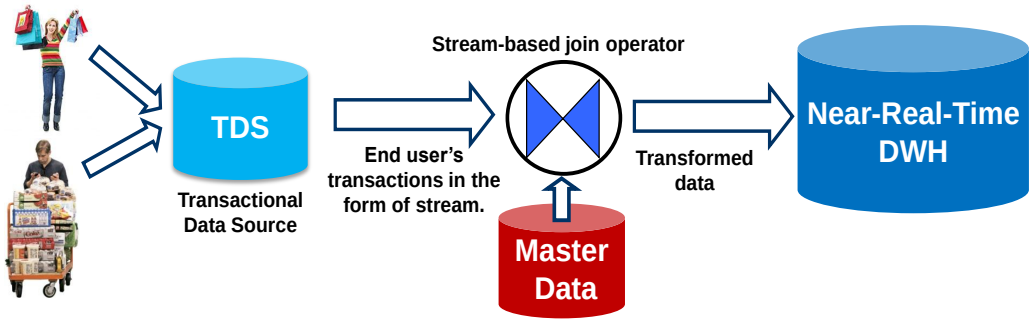


Figure 1.1: An example of stream-based joins in mobile networks

data before being propagated in the data warehouse.

Organisations that become more environmentally aware will try to reduce the carbon footprint of their IT infrastructure. A main-memory approach can be power-hungry, as can cloud-computing approaches. Therefore, approaches that can work with limited main memory are of interest. This does not mean that they are always optimal under all resource parameters, but they can serve as an option in a portfolio of building blocks for a resource-aware system-setup.

Another important aspect we can consider in Figure 1.1 is the nature of the end user's data. Normally this is a non-uniform distribution with a certain value of skew parameter. The popular types of distributions in this context are Zipfian distributions. In the literature these Zipfian distributions are discussed as an acceptable model for sales, where some products are sold frequently while most are sold rarely [3]. Zipf's Law is the special case where the exponent value is equal to 1. Another important case is the 80/20 Rule. This corresponds to a different exponent value namely 0.8614 [66].

1.5 Problem Statement

In the context when a stream is joined with persistent data, one of the significant factors for choosing the join algorithm is that the inputs in the join come from different sources with different arrival rates. The stream input is fast, high volume and has a bursty nature while the access rate of persistent data is comparatively slow due to the disk I/O cost; therefore, a bottleneck is created during the join execution. The challenge in this case is to eliminate this bottleneck by amortising the expensive disk I/O cost over a high volume of stream data.

One well-known approach, MESHJOIN (Mesh Join) [90, 91] has been proposed in the literature for processing stream data with persistent data, particularly for the scenario of data warehousing. The main objective of this approach is to amortise the expensive disk I/O cost over a high volume of stream data. Although it is a useful attempt in this direction, it leaves some issues open for research. First, due to complex dependencies among the components of MESHJOIN the algorithm cannot attain maximum performance (details are available in Chapter 4). Secondly, common market characteristics like Zipfian distribution for stream data and intermittency were not taken into account when the algorithm was designed. As a consequence MESHJOIN cannot deal with stream intermittency effectively and also in MESHJOIN the performance is inversely proportional to the size of the master data (details are presented in Chapter 5).

These motivations have led us to design an efficient algorithm for processing non-uniform stream data which is a common characteristic of real world applications.

1.6 Thesis Contributions

The primary aim of this thesis is to design a robust stream-based join algorithm for processing stream data with persistent data. While achieving this aim the thesis also provides the following contributions.

1. It provides a literature review of stream-based join algorithms. These join algorithms are classified into two categories based on the nature of their input parameters. Each category analyses a number of related approaches with their strengths and weaknesses.

2. The detailed literature review has motivated an investigation of a well-known stream-based algorithm called MESHJOIN. This has identified a number of research issues in the MESHJOIN algorithm. MESHJOIN provides a small amount of data on the position of the optimal value depending on the memory size, and no performance comparison has been carried out between the optimal and reasonable default sizes for a key component called disk-buffer. In this research more details are provided about the optimal and default settings for MESHJOIN.

3. MESHJOIN distributes memory among its components suboptimally due to some complex dependencies. An alternative approach called R-MESHJOIN (reduced Mesh Join) is presented here. R-MESHJOIN removes these complex dependencies and divides memory among the components optimally.

4. MESHJOIN uses a non-adaptive strategy to access the master data and therefore cannot deal with the intermittency in stream data efficiently. A new algorithm HYBRIDJOIN (Hybrid Join), is introduced to handle both of these issues. A synthetic workload is also generated to test the performance of HYBRIDJOIN.

5. HYBRIDJOIN does not consider non-uniform distributions such as Zipfian distribution, that can affect the stream data. The stream data is analysed and an appropriate algorithm called X-HYBRIDJOIN (Extended Hybrid Join) has been designed for processing the non-uniform stream data. Further, X-HYBRIDJOIN has been improved in the form of Optimised X-HYBRIDJOIN by using efficient and more appropriate data structures. Optimised X-HYBRIDJOIN assumes the master data is sorted in the order of access frequency. Under this assumption the algorithm can perform better than all the other approaches described.

6. Finally, this thesis introduces a robust stream-based algorithm called CACHEJOIN (Cache Join). CACHEJOIN has the capability of dealing with unsorted master data efficiently. The algorithm performs significantly better than all other approaches on unsorted master data while a little less well than Optimised X-HYBRIDJOIN on sorted data.

The aim of this thesis is accomplished by providing a choice of algorithms depending on the nature of stream input data and the master data being processed. R-MESHJOIN can perform efficiently when stream input data is uniform and continuous regardless of whether the master data is sorted or unsorted . Optimised X-HYBRIDJOIN will perform better when stream input data is non-uniform and the master data has been sorted. Finally, CACHEJOIN is more appropriate for joining non-uniform stream input data with unsorted master data.

1.7 Scope of the Thesis

The scope of the thesis can be defined in two dimensions. One dimension includes the characteristics of stream data whereas the other dimension covers the characteristics of master data. On stream data two common characteristics are considered here. The first characteristic is non-uniform stream data, which can be found in most real world applications. For example in supermarkets only a few products among all those available are sold frequently [3]. The second characteristic is related to the flow of stream data. Again, if we consider the same example of the supermarket, the rate of transactions in peak hours is much higher than in less busy hours. A number of applications where data streams have both of these characteristics can be found in literature [32, 63, 65, 67, 74, 89, 116, 117]. With respect to disk-based master data, the characteristics of indexing are important here.

The coverage of the approaches proposed with respect to the specified dimensions is shown in Figure 1.2. CACHEJOIN is a robust algorithm that can deal with stream data, having both characteristics, without any assumption about ordering the master data. The two other algorithms, X-HYBRIDJOIN and Optimised X-HYBRIDJOIN, can also deal with stream data having both characteristics. However, both these algorithms assume the master data has been sorted by access frequency. For uniform and intermittent stream data HYBRIDJOIN is a more appropriate algorithm. HYBRIDJOIN can work for both sorted and unsorted master data. Neither MESHJOIN nor R-MESHJOIN has any assumption of an index on the master data and they work effectively only for uniform and continuous stream data.

The following are some possible extensions that are not covered in this research.

- This research limits its focus to join algorithms which keep resource consumption low and therefore does not consider those scenarios where a large amount of memory is available for join processing.

- This research considers unique tuples in master data and an equijoin is applied between key values in master data with foreign key values in stream data. The possibility of non-equijoin is not covered in this research.

- The major focus of this thesis is on non-uniform stream data and therefore it does not consider categorical attributes in master data, e.g. it does not consider equijoins

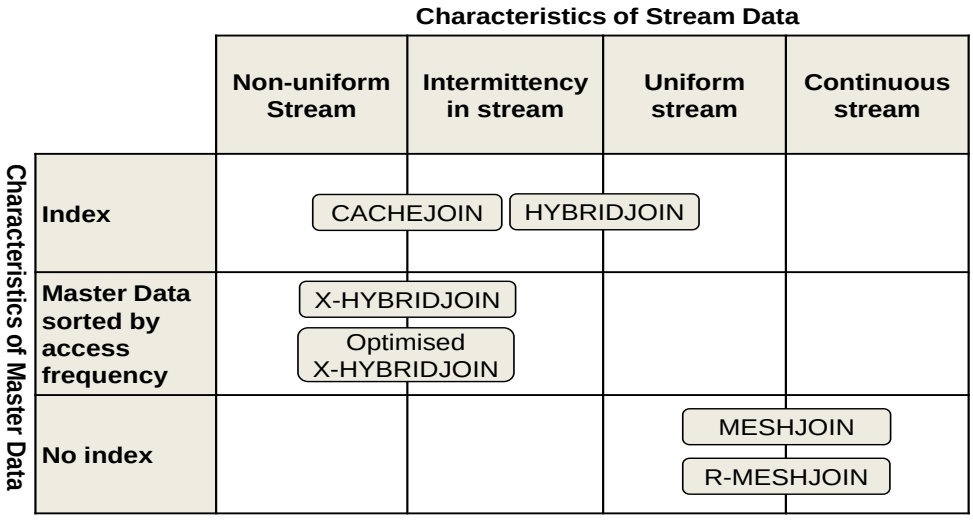


Figure 1.2: Thesis coverage with respect to stream and master data characteristics

over attributes such as gender.

1.8 Structure of the Thesis

Chapter 2 presents a literature review of research on stream-based join operators. For clarity these stream-based join operators are classified into two categories based on several characteristics. Under each category a number of stream-based operators are described, along with their limitations.

Chapter 3 provides the background of the research by exploring MESHJOIN and highlights the important issues that have been investigated in developing MESHJOIN. This chapter also addresses the issue of memory setting for a key component of MESHJOIN.

Chapter 4 addresses the second issue of MESHJOIN, which is the unnecessary dependency between the join components. A revised version of MESHJOIN called R-MESHJOIN is presented. R-MESHJOIN removes the complex dependencies that create the problem in MESHJOIN and a simple and accurate cost model for memory distribution among the components is introduced.

Chapter 5 deals with two further issues in the existing MESHJOIN algorithm. One is that it uses an inefficient approach to accessing the master data whereas the other one is that it does not deal efficiently with intermittency in stream data. To resolve these issues a new join algorithm called HYBRIDJOIN is introduced here. The cost model is

derived for HYBRIDJOIN and the algorithm is tuned on the basis of that cost model. A synthetic data set is generated for testing the performance of the algorithm.

Chapter 6 presents an extended version of HYBRIDJOIN, called X-HYBRIDJOIN. X-HYBRIDJOIN is designed to take the non-uniform characteristics of stream data into account. X-HYBRIDJOIN contains all the characteristics of HYBRIDJOIN plus an additional feature of caching the most frequently-used part of the disk data permanently in memory. This additional feature adds a significant contribution in the algorithm's performance, especially for non-uniform distributions as found in real world applications. X-HYBRIDJOIN is also tuned to its optimal settings using both empirical and mathematical tuning approaches.

Chapter 7 discusses further improvements to X-HYBRIDJOIN. The X-HYBRIDJOIN algorithm uses two buffers for loading the master data. One buffer is non-swappable whereas the other one is swappable. However, the algorithm treats both buffers in the same way; further it does not use an efficient data structure for the non-swappable buffer. This chapter discusses some modifications in X-HYBRIDJOIN and presents an alternative in the form of Optimised X-HYBRIDJOIN. Optimised X-HYBRIDJOIN treats both buffers independently, using efficient data structures that eventually improve the performance of the algorithm.

Chapter 8 describes a generalised approach for processing stream data with master data. Although Optimised X-HYBRIDJOIN performs optimally for non-uniform distributions, the algorithm includes the assumption that the master data are sorted with respect to access frequency. In this chapter this assumption is removed and a robust algorithm called CACHEJOIN is introduced. Experiments prove that for non-uniform distributions CACHEJOIN performs optimally for all the other algorithms except Optimised X-HYBRIDJOIN when the master data is unsorted. The cost model is also derived for CACHEJOIN and the algorithm is tuned empirically and mathematically.

Chapter 9 concludes this research by summarising the contributions and providing some future guidelines.

2

Related Work

2.1 Introduction

A stream-based join is an operator used for combining or interrelating various streams or a stream with master data. Chapter 1 listed a number of application scenarios where stream-based joins play an important role. One question that arises here is why these stream-based joins are important given that traditional joins already exist and are well understood. The answer is that in a stream setting the tuples arrive continuously and they usually need to be processed in the same fashion. In addition to that it is not possible to store the entire stream in memory and hence one cannot apply the indexing feature to extract the tuples. Therefore, the traditional blocking join operators [18, 34, 35, 36, 55, 64, 70, 71, 72, 93, 94, 114, 118] will no longer work for these settings.

To process the continuous stream data non-blocking pipelined join operators [38, 41, 56, 57, 75, 79, 100, 105, 113] are required. Normally these joins are stateful operators. Otherwise they need to keep a record of all past stream tuples and that will ultimately

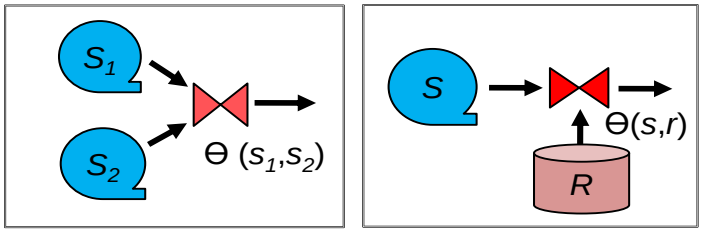
outgrow the available memory [111]. Therefore, one important challenge for non-blocking join operators is join state management. One possible solution for this challenge is the concept of a sliding-window. In the sliding-window concept the scope of the join operator is restricted to a recent window. There are two further categories within the sliding-window concept. One is a time-based sliding-window in which the scope of the join operator is restricted to a particular time period. In this case a new tuple of stream A can join with a set of tuples of stream B that arrived in the last specified time interval. The other category is a tuple-based sliding-window in which each new tuple of stream A can join with n recently arrived tuples of stream B. With both the time-based sliding-window and the tuple-based sliding-window the slide is forwarded after either a particular time or a specified number of tuples [111]. More details about the semantics of sliding-window are presented in Continuous Query Language (CQL) [8].

2.2 Basic Characteristics of Stream-based Joins

A large amount of research has been carried out already in the area of stream-based joins. The focus of this research is on a particular group of stream-based joins called hash-based joins. Based on input characteristics these hash-based joins can be classified further into two categories. The first category includes all those join approaches in which both join inputs are in the form of streams and later in this chapter the term $Stream - Stream$ is used for this category. The second category contains those join approaches in which one input is in the form of a stream whereas the other is disk-based. This category is represented in this chapter by the term $Stream - Disk$. The following subsections present the basic semantics of joins for each category.

2.2.1 Semantics of Join for Stream-Stream

As described earlier a stream is a combination of an infinite number of tuples of the form $< s, t >$ where s is a tuple and t is a timestamp for that tuple. The two streams S_1 and S_2 are joined together by maintaining their join states separately as shown in Figure 2.1(a). A join state of S_1 keeps the record of tuples of stream S_1. Similarly the join state of S_2 keeps the record of received tuples of stream S_2. When the tuple $s_1 \in S_1$ arrives, it is



(a) Both inputs are in the form of stream

(b) One input is stream while the other is disk-based

Figure 2.1: Graphical representation of stream-based joins

stored in the join state table for S_1 and probed into the join state table of S_2 and join output is then generated. Similar operations are performed for stream S_2.

2.2.2 Semantics of Join for Stream-Disk

In this type of join each stream tuple $s \in S$ is joined with the disk-based tuple $r \in R$ [8, 52] as shown in Figure 2.1(b). Normally the disk-based relation R is stored in a database. The main issue with these types of joins is that the disk I/O cost for accessing the database is a dominant factor. The stream arrival rate is fast and therefore, it is necessary to amortise the dominant I/O cost over the fast stream. The focus of this dissertation is to deal with such kinds of stream-based join operators. The following sections describe the relevant join operators under each category.

2.3 Stream-Stream

A number of well known stream-based join approaches are reviewed which take both inputs in the form of a stream. However the target of this research is to process stream data with master data stored on disk. The reason for considering the join approaches under the Stream-Stream category in our literature review is to introduce the common hash-based architecture used in all these approaches. The strengths and weaknesses of each approach are also highlighted.

2.3.1 Symmetric Hash Join (SHJ)

Symmetric Hash Join (SHJ) [108, 109] has exploited the concepts of the traditional hash join algorithms by eliminating the delay for the input streams. SHJ maintains the hash

tables for both input streams in memory and the algorithm assumes that both inputs can be held in memory in their entirety. Each new tuple from one stream is joined with the tuples of the other stream, stored in the hash table, and the output for the joined tuple is then generated. After the output has been generated, that new tuple is stored in its own hash table. The algorithm can generate the output as soon as the input tuple from either stream arrives. However it needs to store both inputs in memory.

2.3.2 Double Pipelined Hash Join (DPHJ)

The double Pipelined Hash Join (DPHJ) [59] is an extension of SHJ and can produce the output faster than SHJ. The algorithm stores each input in a separate hash table. If the hash table becomes filled the algorithm flushes the additional tuples to disk to process them later. The algorithm processes the disk-resident tuples after finishing both inputs. DPHJ removes one limitation of SHJ that of storing the entire join state in memory. DPHJ uses a flag strategy to avoid producing the duplicate tuples. This algorithm is suitable for medium size data and does not perform well for large size data.

2.3.3 XJoin

XJoin [41, 103, 104] is an extended form of SHJ that handles memory overflow by flushing the largest single partition to disk. XJoin presents a three-stage strategy for switching its execution state between disk and memory.

First Stage: The first stage of XJoin is quite similar to the standard Symmetric Hash Join with only one difference; the tuples are organised in partitions both in memory and on disk. Each partition has two parts. One part exists in memory while the other part exists on disk. On the arrival of an input tuple from Source S_1 the algorithm stores the tuple in the relative memory partition and probes it into the corresponding partition of Source S_2. If sufficient space is not available in memory then the algorithm chooses one partition, flushes all its tuples to the disk and resumes the usual process. The first stage continues until the tuples are received from at least one input. If the algorithm does not receive any tuple from either input up to a particular time period then it blocks the first stage and starts the execution of the second stage. The first stage is terminated permanently when both inputs have finished.

Second Stage: The second stage starts its execution when one or both inputs become blocked. In the second stage the algorithm selects a partition, reads the tuples from the disk-resident part of that partition and probes them into the corresponding memory partition of the other source. In the case of a match the algorithm generates the resulting tuple as an output. When all the tuples of the selected disk-resident partition have been processed, the algorithm checks the status of the input stream. If any stream resumes producing tuples the algorithm switches back to the first stage. Otherwise it continues the second stage by selecting a different disk-resident part.

Third Stage: The third stage is also called the clean-up stage and it starts when both inputs have finished. The main objective of this stage is to make sure that all the tuples of both inputs are produced as an output.

XJoin uses a timestamp approach to avoid duplicates. Each tuple is assigned an arrival time when it is loaded into memory and a departure time when it flushes to the disk. The overlapping of timestamps for any two tuples indicates the duplication of a tuple. However, in XJoin the strategy for detecting the duplicate tuples is not effective and some flushing policy is also required in order to transfer the extra tuples to the disk.

2.3.4 Hash-Merge Join (HMJ)

Hash-Merge Join (HMJ) [79] belongs to the series of symmetric joins. It consists of two phases, hashing and merging. The algorithm begins with the hashing phase. During this phase the algorithm reads the input tuples from two data streams and loads them into hash-buckets in memory. The algorithm then joins these hashed tuples and generates the output accordingly. When the memory is filled, the algorithm flushes parts of the hash table to the disk. The second, merging phase of the algorithm starts when both input streams are blocked. During this phase the disk-resident tuples are joined together. Once any source becomes unblocked the algorithm switches to the first phase again. This switching between two phases continues until all the data has been processed. HMJ uses an effective flushing strategy that helps to keep a balanced memory distribution between the two inputs. Moreover, as the algorithm flushes a pair of partitions (one from each source) to disk, no timestamp is required to avoid the duplicate tuples. However, in each flushing the sorting of both partitions is required.

2.3.5 Early Hash Join (EHJ)

In all the join operators outlined above the flushing policy aims to optimise the generation of output. They do not consider minimization of execution time, which is equally important. MJoin addressed this issue but the experimental analysis for calculating the benefit is not presented.

Early Hash Join (EHJ) [73] is an improved version of XJoin that considers the factor of execution time. In EHJ a biased flushing policy is used to flush the data on the disk and a simplified technique is presented to identify the duplicate tuples. EHJ flushes the partition with large input first which is similar to the strategy presented in Dynamic Hash Join [33]. The technique used in EHJ to determine duplicate tuples is based on cardinality. For one-to-one and one-to-many relationships the algorithm does not use any timestamp while for many-to-many relationship it requires an arrival timestamp only.

EHJ follows the concepts of Symmetric Hash Join (SHJ). Each input is stored in a separate hash table. When a tuple arrives from one input, it is probed into the hash table of the other input and, if the tuple matches, then that tuple is generated as an output. After generating the output the tuple is added into its own hash table. The default reading strategy for the in-memory phase is alternative, i.e. the tuples from both inputs are read alternately. However at any time a user can change the reading strategy to optimise the output rate. In the case when the memory gets full the algorithm switches to the second phase, called the flushing phase. The algorithm implements the biased flushing policy that is based on two given rules: (a) the algorithm first selects the largest non-frozen partition of the bigger relation for flushing, (b) if no such partition is found the algorithm selects the smallest non-frozen partition of the smaller relation. When both inputs are finished the algorithm starts its last phase, the clean-up phase, and processes the tuples which were missed in the first two phases.

2.4 Stream-Disk

All join algorithms under this category take one input in the form of a stream while the other input is invoked from the disk. The join processing for such a setup, where the two inputs arrive with different rates, is more challenging than that where all inputs have

similar arrival rates. In the following a selection of join algorithms are discussed, along with their limitations.

2.4.1 Index-Nested-Loop Join

Index-Nested-Loop Join (INLJ) [92] is a traditional join algorithm that can be used to join the stream data with the master data. In INLJ, a stream S is scanned tuple by tuple and the look-up relation R is accessed using a cluster-based index on the join attribute. Although this join algorithm can deal with a bursty stream, it requires extra time to maintain an index on the join attribute. Further it processes one tuple at a time, reducing the throughput.

2.4.2 MESHJOIN

The Mesh Join (MESHJOIN) algorithm [90, 91] has been introduced for scenarios like real-time data warehouses. One input of the join is end-user updates that come in the form of a stream, while the other input is master data that resides on disk and is accessed during the join execution. The key objective of this join is to amortise the fast stream of updates with the slow disk access rate. To achieve this objective the algorithm keeps a number of chunks of stream in memory at the same time. In MESHJOIN, the master data is traversed cyclically in an endless loop and every stream tuple is compared with every tuple in the master data. Therefore every stream tuple stays in memory for the time that is needed to run once through the entire master data.

MESHJOIN is an adaptive algorithm with respect to stream amortising, there are some research issues such as inefficient memory distribution among the join components, ineffective strategy to access the master data, and inability to deal with intermittency in the data streams that need to be discussed further. More details about these issues are presented in Chapter 3.

2.4.3 Partition-based Approach

A partition-based approach [21] has been introduced to deal with intermittency in the stream. Similar to MESHJOIN, this partitioned-based algorithm also divides the master data R into segments using a space-partitioning technique. A subset of these segments

resides in memory. A wait-buffer is another memory-based component that contains parallel slots which are equal in number to the disk segments. For each stream update tuple, the join is performed if the required disk tuple is available in memory. Otherwise the stream tuple is mapped into the corresponding slot in the wait-buffer. The disk-invokes operations based on the following conditions: (i) if the number of stream tuples in any slot of the wait-buffer crosses a threshold value or (ii) if the memory space allocated for the wait-buffer is full. In the case of the first condition the algorithm loads a particular disk segment into memory (R-buffer) and a join is executed between the tuples residing in that particular slot of the wait-buffer and the tuples in the disk segment. In the case of the second condition the algorithm invokes the disk segments one by one according to the order of the sizes of the slots in the wait-buffer and performs a join between the stream tuples and the disk tuples. The disk-retrieved segment may also replace an in-memory disk segment, depending on the frequency of the retrieved segment tuples in the arrival stream.

A general observation is that the join attribute values waiting in the slots of the wait-buffer, which are not frequent in the input stream, need to wait even longer than in the original MESHJOIN algorithm, where the slot does not reach the threshold limit. In addition the author of the partition-based approach focuses on the analysis of the stream buffer in terms of back log tuples and the delay time rather than analysing the performance of the algorithm. The cost model is not provided for the approach. Also, the algorithm requires a clustered index or an equivalent sorting on the join attribute and it does not prevent starvation of stream tuples.

Another join operator called Adaptive, Hash-partitioned Exact Window Join (AH-EWJ) [22] has been introduced to produce accurate results for sliding window joins over data streams. This approach can also be used in the scenario where a stream joins with master data. However, the focus of this approach is on the accuracy of the join output rather than on performance optimisation while considering non-uniform characteristic on the stream data.

A number of tools have been developed for stream warehousing which can also process stream data with master data [13, 46, 47, 48, 49, 50]. However, these tools do not provide optimal solutions for non-uniform characteristic of real world data.

Some other approaches [23, 24, 69] have also considered the problem of joining stream

data with master data, but to the best of our knowledge they did not propose any appropriate algorithm for it.

3

Analysis of MESHJOIN

3.1 Introduction

The MESHJOIN algorithm was introduced [90, 91] to perform joins between a continuous stream and a disk-based relation, we called it master data, using limited memory. One example of such a scenario is near-real-time data warehousing where the source data needs to be transformed to a target format during the transformation phase of ETL (Extract, Transform, Load). One input of the join is received in the form of a stream and consists of updates performed on the data source. The other input is a table that is stored on a disk and typically represents slowly-changing master data. The bottleneck occurs while accessing the master data; therefore the key challenge is to amortise the slow disk access cost over a fast data stream. MESHJOIN is an algorithm that addresses this issue. It joins a fast stream S with a large master data R under a limited memory budget. The algorithm can be tuned to perform optimally for a given memory size or to minimize the memory usage required for a given service rate. The term service rate or throughput is

defined as the total number of stream input tuples processed in unit of time.

This chapter first explains MESHJOIN with respect to its components and basic operation and goes on to present certain observations about this algorithm. It also evaluates the MESHJOIN algorithm and proposes default settings in order to remove the tuning effort that MESHJOIN goes through for every new setting. The performance of the algorithm is compared using both optimal and default settings. The results presented at the end of this chapter show that under default settings the algorithm performs only minimally less efficiently (under two percent) than the optimal settings.

In detail, the chapter is structured as follows. Section 3.2 focuses on the working, architecture and algorithm for MESHJOIN. Observations about MESHJOIN are discussed in Section 3.3 while the further investigations of it are presented in Sections 3.4 and 3.5. Tuning and performance comparisons using default and optimal values for the disk buffer sizes are presented in Section 3.6. Section 3.7 explains the strategy for choosing the default value for the size of the disk buffer. Finally Section 3.8 presents a summary of the chapter.

3.2 MESHJOIN

Overall, the MESHJOIN algorithm is a hash join, where the stream serves as the build input and the master data serves as the probe input. A characteristic of MESHJOIN is that it performs a staggered execution of the hash table build in order to load stream tuples more steadily. In MESHJOIN, the whole R is traversed cyclically in an endless loop and every stream tuple is compared with every tuple in R. Therefore every stream tuple stays in memory for the time that is needed to run once through R. The stream tuples that arrive later start the comparison with R from a later point in R (except for the case that the traversal of R resets to the start of R) and wait until this point is reached again in the cyclic reading of R. The chunks of stream therefore leave main memory in the order in which they enter main memory and their time of residence in main memory is overlapping. This leads to the staggered processing pattern of MESHJOIN. In main memory the incoming stream data is organised in a queue, each chunk of stream defining one partition of the queue. At each point in time, each partition has seen a larger number of iterations than the previous.

This section elaborates on the major components and the workings of the MESHJOIN algorithm. In order to get a better understanding the pseudo-code of the algorithm is also presented in this section.

3.2.1 MESHJOIN Components

The pictorial representation of MESHJOIN components is shown in Figure 3.1. The following explains each component, along with its functionality.

Disk buffer: The disk buffer is a key component in MESHJOIN which is used to load the master data into memory in the form of partitions, but does so one partition at a time. This is important because the disk I/O cost is directly related to the size of this component. In MESHJOIN changing the size of the disk buffer also changes the number of partitions in the queue.

Hash table (H): To amortise the disk I/O cost MESHJOIN keeps a number of stream tuples in memory at a time. The algorithm loads these stream tuples in chunks which can be differentiated with respect to their loading time. The hash table is the component used to store these stream tuples. During execution, disk tuples stored in the disk buffer are probed in the hash table and output is generated if the probing is successful.

Queue (Q): The queue is another component in MESHJOIN that stores the join attribute values for stream tuples. The main purpose of the queue is to keep a record of the oldest tuples. The oldest tuples are the tuples that have been joined with the whole R; in the next step they have to be deleted from memory. The relationship between the memory size of the queue and the memory size of the hash table is linear.

Stream buffer: The stream buffer is a small component in MESHJOIN. The main purpose of this component is to hold the fast stream for a while until the necessary space is created in the memory.

3.2.2 Basic Operation

As shown in Figure 3.1, there are two input sources, one is a continuous data stream S and the other is master data R. MESHJOIN continuously scans the data from these input sources and joins them together in order to generate the result. The usually large master data R has to be stored on disk, and is read into memory through a disk buffer of size b

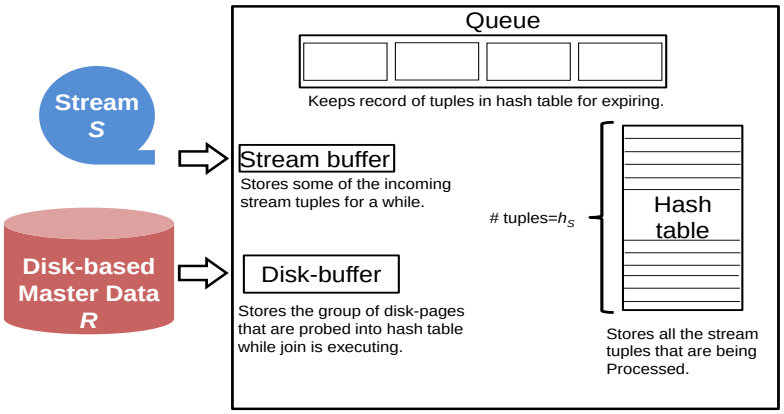


Figure 3.1: MESHJOIN components

pages. The master data R is naturally split into k equal partitions, where each partition is of size b number of pages. One traversal of R therefore happens in k steps. In each step a new partition of R is loaded into the disk buffer and replaces the old content. These steps are referred to as iterations in the following.

The key concept in the execution of the algorithm is that, for each iteration, it loads one partition from disk into the disk buffer and a set of stream tuples into the hash table, while also placing the values of their join attributes in the queue. The tuples stored in the disk buffer are then joined with all the tuples stored in the hash table and the output is generated. At the start of the next iteration the oldest stream tuples are discarded from the hash table together with their join attribute values from the queue. In the next iteration the algorithm again loads the new stream input into the hash table and queue while the disk input is loaded into the disk buffer. The advantage of this algorithm is that it amortises the fast arrival rate of the incoming stream by executing the join of disk pages with a large number of stream tuples. Figure 3.2 shows a pictorial representation of the MESHJOIN operation at the moment that a partition R_2 of R is read into the disk buffer but has not yet been processed. The figure shows that the stream tuples whose join attribute values are stored in partition Q_1 of the queue have already seen all the disk partitions except R_2. Therefore, after the processing of R_2 they will expire while the stream tuples storing the join attribute values in partitions Q_2, Q_3 and Q_4 will move one step forward after the processing of R_2.

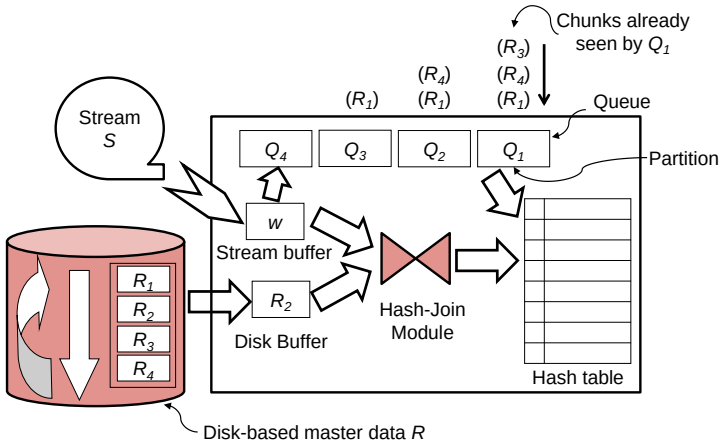


Figure 3.2: MESHJOIN before processing R_2

Algorithm 1 MESHJOIN [90, 91]

Input: A master data R and a Stream S
Output: $S \bowtie R$
Parameters: w tuples of S and b pages of R
Method:

```
1:  while (true) do
2:    READ b pages from R and w tuples from S
3:    if Q is full then
4:      DEQUEUE T from Q where T are w pointers
5:      REMOVE the tuples of hash H that correspond to T
6:    end if
7:    ADD w tuples of S in H
8:    ENQUEUE in Q, w pointers to the above tuples in H
9:    for each tuple r in b pages of R do
10:     Output r ⋈ H
11:   end for
12: end while
```

3.2.3 Algorithm

The pseudo-code for the original algorithm is shown in Algorithm 1 [90, 91]. For each iteration, the algorithm takes two parameters, w tuples and b pages, from the input sources S and R respectively and feeds them into the relevant buffers. Before starting the join execution the algorithm monitors the status of Q. If it is already full, the algorithm dequeues the pointer addresses of the oldest tuples and discards the corresponding tuples from the hash table. In the next step the algorithm stores w tuples in the hash table with their corresponding addresses into Q. Finally, it generates the required output after performing the join of b pages with all tuples in the hash table.

3.3 Research Issues

Although the MESHJOIN algorithm is an interesting algorithm for processing stream data with persistent data by amortising the slow disk access cost on a fast data stream, there are some research issues that need to be explored further. A brief description of these issues is presented here, while in later chapters they are discussed in more detail.

3.3.1 Settings

The disk buffer is a key component in the MESHJOIN algorithm used to load the disk data into memory. MESHJOIN reserves a variable size of memory for this component and the authors did not evaluate the procedure to measure the size of memory for this component. Moreover, the algorithm tunes this component for every new setting of memory. In this chapter it will become clear that using the default memory setting for this component makes no noticeable decrease in the performance as compared to the optimal memory setting. There are two main reasons behind this performance analysis. First, it is important to know how much the tuning module actually contributes in the performance of the algorithm. Secondly, the analysis provides the user with an option: the acceptance of a small degradation in performance while avoiding having to use the tuning module.

3.3.2 Dependencies

In MESHJOIN there are some unnecessary dependencies between the components of the algorithm. As a result, first the algorithm is slightly suboptimal even after tuning, and second the size of a key component, the disk buffer varies when the size of R on disk is varied, which is counter intuitive. In our investigations these issues can be resolved by discovering the true dependency between the components.

3.3.3 Disk access

MESHJOIN reads R sequentially and there can be a number of partitions in R which do not have a single match with a stream tuple in memory. In such cases the algorithm produces an overhead in the form of disk I/O cost that affects the performance of the algorithm negatively. Moreover, increasing the size of R on disk also increases the time

for which each stream tuple remains in memory.

3.3.4 Intermittency

MESHJOIN cannot deal effectively with intermittency occurring in the stream. The reason for this is that when a pause occurs, the stream tuples that are already in memory will hang for an indefinite period of time.

Issue 3.3.1 is addressed in this chapter. Chapter 4 deals with issue 3.3.2. Finally, issues 3.3.3 and 3.3.4 are addressed in Chapter 5.

3.4 Settings

This chapter discusses an optimisation problem for a critical component of MESHJOIN, the disk buffer. As shown in Figure 3.2, the disk buffer component of MESHJOIN is used to load disk data into memory and its size varies with a change in the total allocated memory for join execution. Therefore, in order to achieve the maximum service rate within a fixed memory budget, MESHJOIN first tunes that disk buffer component. The parameters that MESHJOIN uses in tuning are based on a cost model.

To explore the analytical steps behind this tuning process the cost equations are considered [90, 91], both in terms of memory and processing, used by MESHJOIN. To calculate the memory cost, MESHJOIN uses Equation 3.1 while the symbols used in the cost equations are explained in Table 3.1.

$$M = b \cdot v_P + w \cdot v_S + w\frac{N_R}{b}\text{sizeof}(ptr) + w \cdot f\frac{N_R}{b}v_S \tag{3.1}$$

where M is the total memory reserved by all join components, which can be less than or equal to the maximum memory budget, $b \cdot v_P$ is the piece of memory allocated for the disk buffer, $w \cdot v_S$ is the memory reserved for the stream buffer, $w\frac{N_R}{b}\text{sizeof}(ptr)$ represents the memory reserved by the queue and finally, $w \cdot f\frac{N_R}{b}v_S$ is the memory allocated for the hash table.

MESHJOIN processes w tuples in each iteration of the algorithm. The processing cost

Table 3.1: Notations used in cost calculation of MESHJOIN

Parameter name	Symbol
Size of each tuple of S (bytes)	v_S
Number of pages in R	N_R
Size of each tuple in R (bytes)	v_R
Size of each page in R (bytes)	v_P
Number of pages of R in memory for each iteration	b
Total number of iterations required to bring the whole R into memory	k
Number of stream tuples read into join window for each loop iteration	w
Hash table fudge factor	f
Cost of reading b disk pages into the disk buffer (seconds)	$c_{I/O}(b)$
Cost of removing one tuple from H and Q (seconds)	c_E
Cost of reading one stream tuple into the stream buffer (seconds)	c_S
Cost of appending one tuple into H and Q (seconds)	c_A
Cost of probing one tuple into H (seconds)	c_H
Cost to generate the output for one tuple (seconds)	c_O
Total cost for one loop iteration of MESHJOIN (seconds)	c_{loop}
Total memory used by MESHJOIN (bytes)	M
service rate (tuples/second)	μ

for one iteration is denoted by c_{loop} and can be calculated using Equation 3.2.

$$c_{loop} = c_{I/O}(b) + w \cdot c_E + w \cdot c_S + w \cdot c_A + b\frac{v_P}{v_R}c_H + \sigma b\frac{v_P}{v_R}c_O \tag{3.2}$$

where $c_{I/O}(b)$ is the cost to read b pages from the disk, $w \cdot c_E$ is the cost to remove w tuples from the queue and the hash table, $w \cdot c_S$ is the cost to read w tuples from stream S into the stream buffer, $w \cdot c_A$ represents the cost to append w tuples to the queue and the hash table, $b\frac{v_P}{v_R}c_H$ denotes the cost of probing all tuples in b pages into the hash table, and finally, $\sigma b\frac{v_P}{v_R}c_O$ represents the cost of generating output for b pages.
Equation (3.1) can also be written in the following form:

$$w = \frac{M - b \cdot v_P}{v_S + \frac{N_R}{b}\text{sizeof}(ptr) + \frac{N_R}{b}v_S \cdot f} \tag{3.3}$$

Since c_{loop}, as derived from Equation 3.2, is the processing cost for w tuples, the service rate μ can be calculated using Equation 3.4.

$$\mu = \frac{w}{c_{loop}} \tag{3.4}$$

Substituting the value of w in Equation 3.4 gives us the following:

$$\mu = \frac{M - b \cdot v_P}{c_{loop}(v_S + \frac{N_R}{b}\text{sizeof}(ptr) + \frac{N_R}{b} v_S \cdot f)} \tag{3.5}$$

Finding out the maximum service rate depending on b can be done by finding the maximum of Equation 3.5 as a function of b using numerical methods. Numerical methods are necessary, since Equation 3.5 depends on $c_{I/O}$, which is a measured function of b and there is no analytical formula for that. MESHJOIN uses a tuning step, where for each memory budget M, the optimal disk buffer size b is determined by solving this numerical problem. The size of the disk buffer is not fixed and a tuning effort is made for every new memory budget. The issue is whether this tuning effort is really necessary.

The algorithm is evaluated here and an alternative solution is proposed [81][1] to the tuning approach for the MESHJOIN algorithm. The performance of the algorithm is analysed for different sizes of disk buffer, and the performance for the optimal disk buffer size is compared with that for a default size that remains constant for all memory budgets. A difference emerges of less than two percent. In the straightforward implementation of MESHJOIN, the tuning component has full control over the buffer size. Since the tuning component has a sizeable code base, it can include errors. A typical estimate assumes 20 errors per 1000 lines of code [42]. These errors can produce widely-deviating buffer sizes, or worse fatal errors. Widely-deviating buffer sizes create a higher risk than the default size. Therefore the findings suggest that in critical applications the tuning component could be omitted and the default size should be chosen.

3.5 Proposed Investigation

The main concern is that the size of the disk buffer is not fixed and this has an effect on service rate which is difficult to predict. To investigate this problem the MESHJOIN algorithm was re-implemented in order to understand its architecture and analyse its performance.

[1] This work has been published in VLDB Workshop (BIRTE'09).

3.5.1 Understanding the Relationships among the Join Components

As shown in Figure 3.1, the components involved in the MESHJOIN algorithm are the disk buffer, the stream buffer, the queue Q, and the hash table H. In order to distribute the total memory among these components, first a specific part of the total memory is assigned to the disk buffer and then the rest of the memory is assigned among all the other components. The stream buffer component reserves a very small amount of total memory (0.5 MB memory for the stream buffer is sufficient for all the experiments reported on here); therefore it is ignored for the moment. The other component is the queue that is directly connected with the hash table and depends on the hash table linearly with respect to memory consumption. Therefore, to get the optimal join throughput, the actual trade-off for memory distribution is between the disk buffer size and the hash table size plus the queue size. Once the size for the disk buffer has been determined, the memory reserved for the hash table and the queue can be calculated using the following relation.

Hash table and queue in bytes: $M - bv_P$

where M is the total memory, b is the size of the disk buffer in pages, and v_P is the size of each page in the disk buffer.

The hash table and the queue contain equal numbers of entries. The difference is that the hash table stores complete tuples while the queue stores only the pointer for that tuple. Therefore, due to the equal number of entries it creates a linear relationship between the hash table and the queue with respect to memory consumption.

Hash table size in bytes: $H_c \times$total number of stream tuples in memory

where H_c is constant for hash table and its value depends on the size of each stream tuple and a fudge factor for the hash table.

Queue size in bytes: $Q_c \times$total number of stream tuples in memory

where Q_c is constant and its value depends on the pointer size for each tuple.

The number of partitions in the queue is equal to the total number of iterations required to bring the whole of R into memory. That number of partitions in the queue depends inversely on the size of the disk buffer. The formulas of MESHJOIN are more involved because they express the queue size not in bytes or number of tuples but in the number of partitions in the queue, and these partitions change their size. Only knowing

Table 3.2: Relationship amongst the components disk buffer size, hash table size and queue size when the total memory budget is 20MB

Total number of pages in disk buffer	Total number of tuples in hash table	Total number of pointers in queue	Total number of partitions in queue
50	205000	205000	1024
100	201609	201609	512
200	193024	193024	256
400	176384	176384	128

the total number of partitions in the queue makes it difficult to determine the size of the queue unless w, the total number of tuples in each partition, is known. As an example, consider wishing to find out the total number of chairs in a room while knowing the total number of rows but not knowing how many chairs there are in each row. The total number cannot be calculated unless the number of chairs in each row is known. In this example, the number of rows maps to the number of partitions in the queue, while the number of chairs in each row maps to the number of pointers in each partition.

The relationship between the disk buffer size and the stream buffer size is of less significance because of the tiny size of w, and therefore increasing or decreasing stream buffer size does not make a difference. The relationships between the disk buffer size, the hash table size, the queue size and the stream buffer size are shown in Table 3.2.

3.5.2 Empirical Analysis

In order to assess the necessity of the tuning process for the disk buffer component in MESHJOIN, empirical results are needed about how the cost function behaves in a real world scenario, and how much better the performance for the optimal setting is, as compared to reasonable default settings. Since the original code was not available, the problem was investigated by implementing the MESHJOIN algorithm as part of this project, incorporating the same assumptions for the input stream and R as described in Section 3.2. As a preview of the findings in this chapter and to indicate where it is heading, Figure 3.3 shows a sample performance measurement of MESHJOIN for different sizes of the disk buffer within a fixed memory budget. Note that in order to magnify the effect under investigation the y-axis does not start with zero. It is observed that the curve has a pronounced knee [39]. The figure shows that the service rate grows drastically up to the

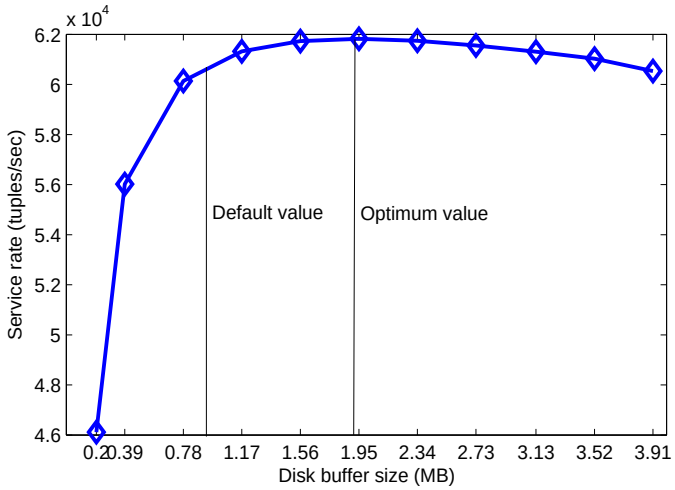


Figure 3.3: Effect of the disk buffer on MESHJOIN performance using fixed memory budget (80MB)

knee in the curve. A saturation behavior can be observed, where incrementing the disk buffer size improves the performance only a little. This is important, because it allows a default value to be chosen which is near the knee of the curve. In the end, a reasonable default value will be determined for the disk buffer size that holds for a series of memory budgets. Before proceeding to the experimental results, we first describe the experimental setup.

3.6 Tuning and Performance Comparisons

At this point the hardware and software specifications for the experiments are described.

3.6.1 Experimental Setup

A prototype of the MESHJOIN algorithm has been implemented using the following specifications.

Hardware specifications: The experiments have been conducted using Pentium-IV machine with 3G main memory. The maximum memory allocated in these experiments is 320MB. The code for the implementations has been written in the Java language. Built-in plug-ins, provided by Apache, and built-in functions like *nanoTime()*, provided by the Java API, have been used to measure the memory and processing time. In addition, the Java hash table does not support the storage of multiple tuples against one key value. To resolve this issue a multi-hash-map, provided by Apache, has been used in the experiments.

Table 3.3: Experimental data characteristics

Parameter	Value
Master data	
Size of R	3.5 millions tuples
Size of each tuple	120 bytes
Default size for the disk buffer	0.93MB
Stream data	
Size of each tuple	20 bytes
Size of each pointer in Q	4 bytes
Fudge factor for the hash table	4.8

Data specifications: The performance of MESHJOIN has been analysed using synthetic data. The look-up data R is stored on disk using a text file format, while the stream data is generated at run time using our designed random-number generator. The experiments have been tested with varying sizes of disk buffer to find its optimal default value. On the other hand, the size of the stream buffer is flexible and fluctuates with the size of the disk buffer. Similarly the size of Q (in terms of partitions) also varies with the total number of iterations required to bring the whole R into the disk buffer. The detailed specification of the data that is used for analysis is shown in Table 3.3.

System of measurement: The performance of the join is measured by calculating the number of tuples processed in a unit second, which is the service rate and is denoted by μ. The measurements are taken after a few iterations of the loop. For increased accuracy three readings are taken for each specification and then their average is considered. Moreover, it is assumed that no other applications run in parallel during the execution of the algorithm.

3.6.2 Tuning of Disk Buffer for Different Memory Budgets

The optimal values of the disk buffer size for a series of memory budgets have been analysed first and then the join performance has been observed at these optimal values. In order to obtain the optimal value for the disk buffer size MESHJOIN has been tuned for a series of memory budgets. Figure 3.4 depicts the optimal values for the disk buffer size in the case of different memory budgets. The figure shows that the size of the disk buffer increases with an increase in the total memory budget. As the total memory M depends on w and b and that w also depends on b, the optimal size of disk buffer b will

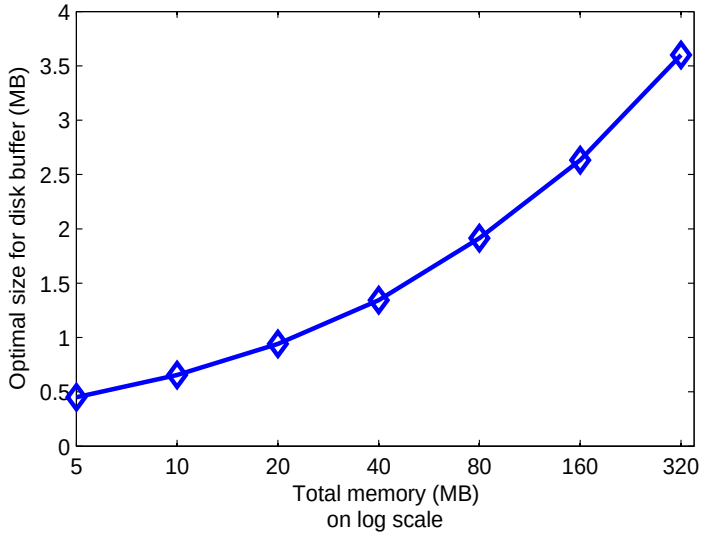


Figure 3.4: Optimal values for the disk buffer size with respect to the different memory budgets

increase with an increase in the total memory budget.

3.6.3 Performance Analysis using Default and Optimal Values for the Disk Buffer Size

In this experiment the MESHJOIN algorithm has been tested for a series of memory budgets in order to observe the real difference in performance for a reasonable default value and optimal values of the disk buffer size. Figure 3.5 shows performance measurements for different memory budgets along with the default and optimal values for the disk buffer size. The optimal value for 20MB is 0.93MB. The setting 20MB is the memory budget from the original MESHJOIN algorithm, and for today's computing landscape a very small value for a server component, even when considering limited memory budgets. For the purposes of this discussion we deemed it most helpful to use the optimal value for this setting as the default value, because if a reasonable performance is obtained for all other memory budgets, there is a strong indication that tuning dependent on the overall memory budget is not necessary.

A clear saturation behavior has been observed for all memory budgets. When 40MB is the total memory budget, the value for the optimal disk buffer size is 1.35MB and the improvement in performance as compared to the default size of the disk buffer is only 0.4%. When considering an 80MB total memory budget, the value for the optimal disk buffer size is 1.91MB, with a 1.17% performance improvement. Finally, when 160MB is

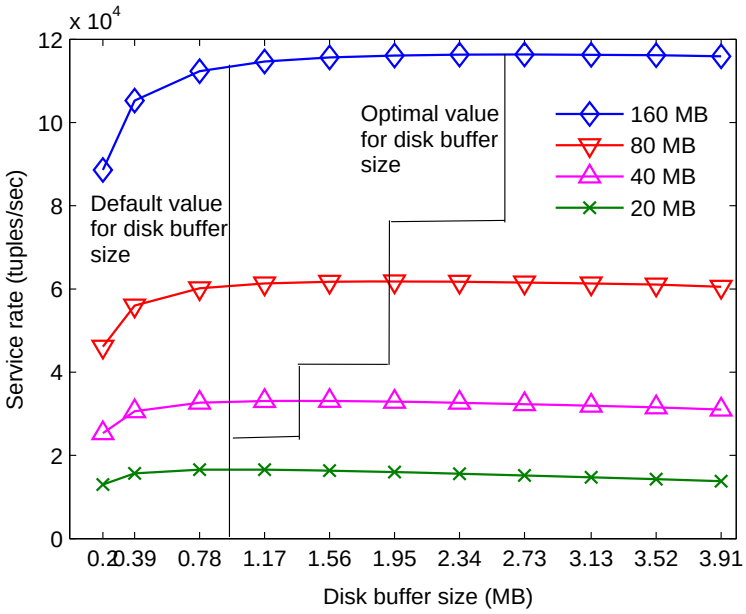


Figure 3.5: Performance comparisons using default and optimal values for the disk buffer size in case of different memory budgets

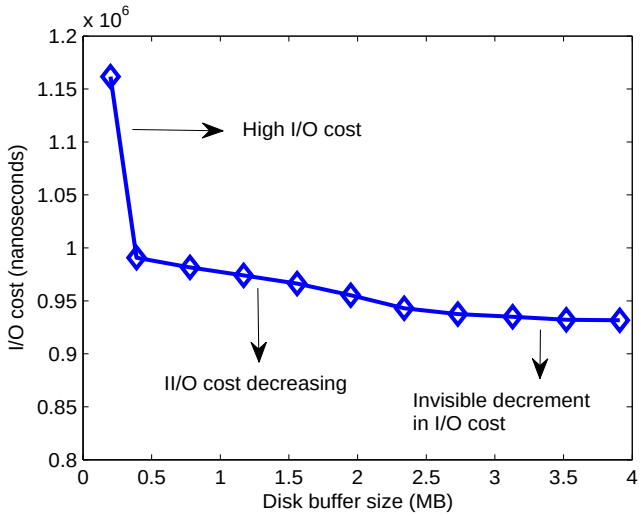


Figure 3.6: Disk I/O cost for different sizes of the disk buffer

the total memory budget, the optimal value for the disk buffer size is 2.63MB, this again improves the performance a little, 1.78%. To prove this experimentally we have measured the I/O cost per page amortised over all pages read into the disk buffer in one iteration. The per page I/O cost for different sizes of disk buffer is depicted in Figure 3.6. The figure shows that in the beginning the I/O cost is high due to the small size of the disk buffer. After that, as the size of the disk buffer increases, the amortised I/O cost per page decreases. But after a while further increments in the size of the disk buffer do not reduce the I/O cost considerably. To visualize the performance difference more clearly,

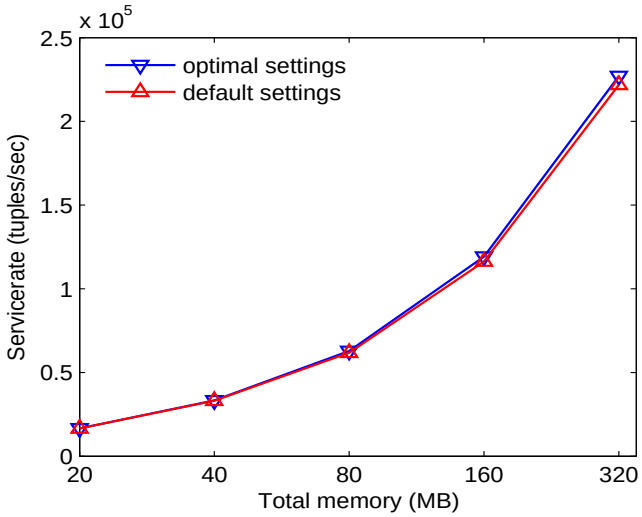


Figure 3.7: Performance comparison directly at default and optimal values of the disk buffer size using different memory budgets

the MESHJOIN performance has been measured directly on the default value and the optimal values of the disk buffer size for a series of memory budgets. Figure 3.7 depicts the experimental results in both cases. It is clear that for small memory budgets the performance of the algorithm is approximately equal, and even with a large memory limit (320MB) there is no remarkable difference in performance.

3.6.4 Cost Validation

In this section the implementation of MESHJOIN has been validated by comparing the calculated cost with the measured cost for different memory budgets. In the case of the calculated cost, the cost for one loop iteration has been calculated using Equation 3.2. The results of this experiment are shown in Figure 3.8. It can be observed from the figure that for every memory budget the measured cost closely resembles the calculated cost, validating the correctness of the MESHJOIN implementation.

3.7 Approach for Choosing the Default Value

Although the function for the performance of MESHJOIN depending on the disk buffer size has a pronounced knee (see Figure 3.3), it is still a smooth curve. Therefore a question arises as to, which exact value should be chosen as a default value. Through practice, it has been observed that a value for the disk buffer size which is optimal for a specific

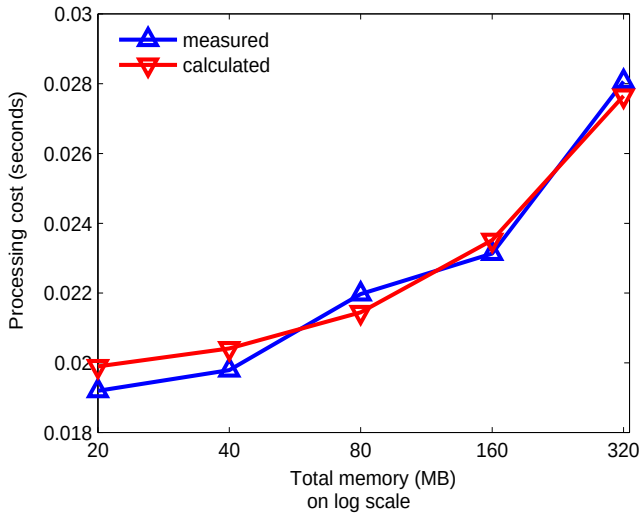


Figure 3.8: Cost validation of MESHJOIN

setting is at the same time a good value for a wide range of settings, and therefore suitable as a default value.

In order to support this a default size of 0.93MB has been chosen in these experiments, which is the optimal disk buffer size for a very small memory budget of 20MB. The experiments have shown that this setting is also sufficient for other memory budgets allocated for MESHJOIN. In particular the results for this default value have been found to be less than 2 percent below the optimal for all tested memory settings. These tests have been restricted to memory sizes up to 320MB. This restriction is motivated by the fact that MESHJOIN, according to the authors of the original publication, is designed for a limited memory budget. In fact the original publications only consider memory budgets up to 40MB, so the investigation up to 320MB has a sufficient security margin. In summary, these experiments have shown that while the optimal disk buffer size varies over a certain range, the performance achieved with them varies only in the order of a few percent. Therefore, in settings where simplicity of the system has precedence over very small performance gains, the default disk buffer size strategy seems worthwhile.

This default value is still dependent on the underlying hardware. Therefore the focus here is primarily on the transferability of default values for settings on the same hardware. Nevertheless it is fair to assume that even across different but similar hardware configurations there will be some transferability.

3.8 Summary

This chapter explores a stream-based join, MESHJOIN. MESHJOIN reserves a variable size of memory for the disk buffer to store R. The procedure for measuring the size of the disk buffer has not been evaluated previously. A further issue is that for every memory setting the algorithm tunes the disk buffer in order to find its optimal value. In this chapter a complete set of parameter settings has been designed for the setup. The example default settings for the setup used here are derived from experimental results. It has been shown here that the default settings are $<2\%$ worse than the optimal settings, which should be taken into account when considering the importance of the optimisation process. Given that the tuning component is a sizeable fraction of the code, and that any code can have bugs, this is an important indication that in mission-critical systems one should consider only using the default setting.

4

R-MESHJOIN

4.1 Introduction

This chapter addresses the issue of unnecessary dependencies between join components in MESHJOIN and presents our solution to resolve the issue. The MESHJOIN algorithm tunes the size of an important component, the disk buffer. However, the tuned MESHJOIN does not provide the best possible solution, due to the complex dependencies in the algorithm. It is further explained in Section 4.2. As a telltale sign, it has been observed that the size of the disk buffer varies with a change in the size of the master data. This is counter-intuitive and raises the suspicion that the memory distribution is not optimal.

These observations have led us to propose a revised version of MESHJOIN called R-MESHJOIN (reduced Mesh Join) [86][1]. The key difference between the two is that R-

[1]This work has been published in the ACM 13th International Workshop on Data Warehousing and OLAP (DOLAP'10).

MESHJOIN has an additional parameter that can vary freely. This removes the complex dependencies that create the problem in MESHJOIN. R-MESHJOIN uses a very simple and accurate cost model which reveals that optimal distribution of the allocated memory requires the real dependency to be between the disk buffer size and the size of the hash table.

In addition, for R-MESHJOIN the optimal size of the disk buffer does not vary with the size of the master data R, thus supporting the intuition that memory distribution in MESHJOIN is not optimal. This can be shown theoretically through the cost model in Section 4.4, and it is verified through experimentation in Section 4.5. In particular the performance of R-MESHJOIN is shown to be slightly better than MESHJOIN.

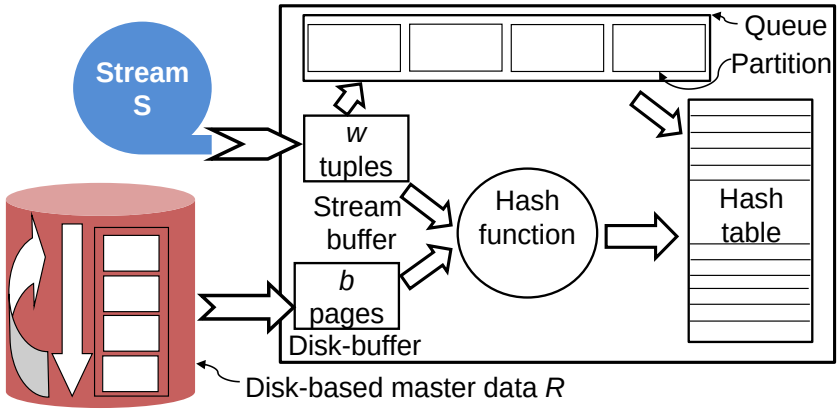
The rest of the chapter is structured as follows. Section 4.2 explains the dependencies between the components of MESHJOIN. Section 4.3 presents the R-MESHJOIN algorithm along with its pseudo-code and also explains the real dependency between the join components. The cost model and tuning for R-MESHJOIN are presented in Section 4.4. The experimental study is described in Section 4.5 and finally Section 4.6 presents a summary of the chapter.

4.2 Dependencies between MESHJOIN Components

A general description of MESHJOIN is presented in Chapter 3. Here MESHJOIN is explored with respect to unnecessary dependencies between the join components.

It has already been stated in Chapter 3 that due to its large size master data R is stored on disk. Since R is loaded into memory using the disk buffer of size b number of pages, MESHJOIN divides R into k partitions while the size of each partition is equal to b pages. To ensure that every stream tuple is joined with the whole of R, the queue also contains k partitions. This is shown in Figure 4.1, where the value of k is equal to four. Increasing the size of R on disk increases the value of k and as a result the size of each partition in the queue decreases. This represents an unnecessary dependency between the memory components in MESHJOIN. Due to this dependency the size of the disk buffer varies when the size of R on disk is varied and therefore, MESHJOIN cannot tune its component optimally. This affects the performance negatively.

To investigate the effect of the size of R on the disk buffer an experiment has been



Size of disk-buffer = one partition = b pages

Iterations required to bring all of R into memory = k (in this example k=4)

tuples in stream buffer = # pointers in one queue partition= $w = \frac{h_s}{k}$

Figure 4.1: Unnecessary dependencies between MESHJOIN components

conducted in which the total memory budget (100MB) for join execution was fixed and the disk buffer has been tuned by varying the size of R. The results of this experiment, shown in Figure 4.2, confirm that in MESHJOIN the optimal value for the disk buffer varies with a change in the size of the master data R. It has also been confirmed that this is the prediction of the MESHJOIN cost model; so it is a real effect. However, this seems to be implausible because of the following argument. Let it assumes that R is replaced by a table R' that is composed of two identical copies of the old R on disk, one after the other, as shown in Figure 4.3. For every given tuning setting, the output and behavior of MESHJOIN would be identical if the two experiments were run side by side, one using R and one using R'. The only difference is that in the experiment using R', whenever MESHJOIN reads a partition R_3 of the second copy of R in R', it would read the original R_3 in the experiment using R. The optimal tuning setting of the disk buffer should therefore be the same. But MESHJOIN's tuning approach gives two different settings for both R and R', and therefore one of them is not optimal. R-MESHJOIN's tuning approach, in contrast, will give the same tuning settings for both table sizes. This research project has also investigated whether it is possible to refine the MESHJOIN algorithm in such a way that the disk buffer size can be kept independent of the size of R. In the following it is shown that this is possible and that it also improves the service rate for certain sizes of R.

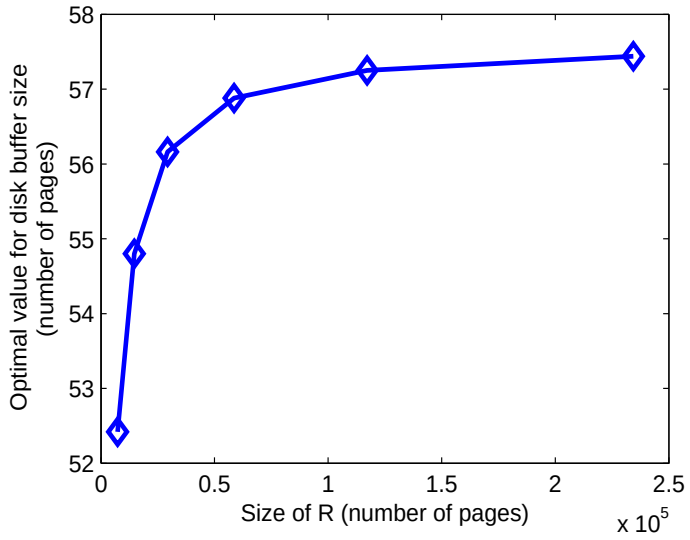


Figure 4.2: Effect of R on the tuning of the disk buffer

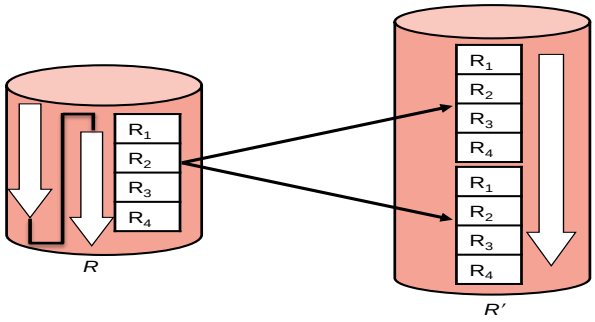


Figure 4.3: Illustration for argument concerning tuning approach and master data size

In summary, MESHJOIN tunes its disk buffer size to optimise the performance within the available memory resources. The tuning process is based on a dependency between the number of iterations required to bring the master data into the memory and the size of partitions in the stream queue. Due to this unnecessary dependency the algorithm cannot tune its components optimally and as a result the algorithm cannot perform with its maximum efficiency.

4.3 R-MESHJOIN

A modified version of MESHJOIN called R-MESHJOIN (reduced MESHJOIN) has been proposed here. In R-MESHJOIN the size of the disk buffer is not affected by changes in the size of the master data R. The data structures and execution architecture for the proposed R-MESHJOIN are shown in Figure 4.4. In the proposed R-MESHJOIN algorithm, the

disk buffer is divided into a number l of logical partitions, each of size b_P pages. After completing the probing of each logical partition, new w tuples are scanned from the stream S into the stream buffer and loaded into the hash table, while enqueuing their pointer addresses in the queue. The disk pages are read after completing the probing of the whole disk buffer. The key purpose of dividing the disk buffer into further l logical partitions is to remove the unnecessary dependency between the numbers of queue partitions and k. The number l varies when the size of R changes. R-MESHJOIN works like MESHJOIN if the value of parameter l is set equal to one (i.e. b_P=b). Normally, l would be greater than one (i.e. b_P <b). In R-MESHJOIN, as opposed to MESHJOIN, it can react to changes in the size of R with changes in l to achieve optimal performance. This means that the total number of partitions in the queue Q_p does not have to change with changes in R. The possibility of varying l provides another degree of freedom. Therefore, R-MESHJOIN has one degree of freedom more than MESHJOIN for the tuning process. The possibility to vary Q_p still exists in R-MESHJOIN. Therefore the tuning process in R-MESHJOIN can vary this parameter if it is necessary. Later experiments will show that the optimal performance is always achieved with the same value for this parameter. Such an outcome shows that the ability to vary this parameter does not contribute to achieving optimal performance, if enough other tunable parameters are available, as in R-MESHJOIN. In MESHJOIN, the parameter l, which, as is known now, has to change to achieve optimal performance, is not available. MESHJOIN can therefore not reach optimal performance by varying l and this explains why MESHJOIN varies Q_p, as the only alternative parameter that can alter the performance, albeit with negative consequences for other parameters.

As in MESHJOIN, in R-MESHJOIN it is also assumed that the input stream S is continuous with a constant arrival rate, and no physical characteristic is considered for master data R. The performance-related assumptions made here about R-MESHJOIN types of join algorithms are as follows:

- If the memory available for join execution increases, the service rate μ will also increase accordingly therefore, $\mu \propto M$.
- The size of R affects the service rate μ because increasing the time to cover R increases the time that each stream tuple needs to stay in the join window therefore, $\mu \propto \frac{1}{R}$.

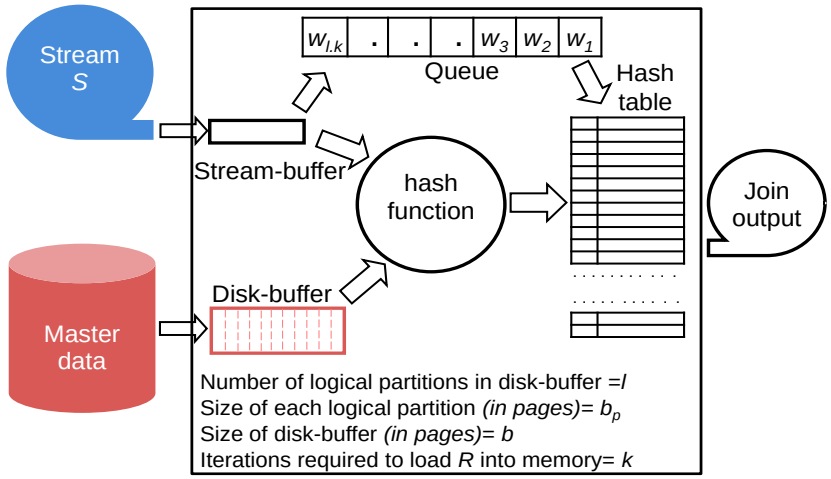


Figure 4.4: Data structures used by R-MESHJOIN

4.3.1 Algorithm

In R-MESHJOIN, since the disk buffer is divided further into equal-sized logical partitions and each partition is denoted by b_P, for each iteration of the algorithm only a fixed and small part, b_P, of b is probed in the hash table. To keep a record of the total number of processed logical partitions we introduce a variable *probedPages* that increases by b_P on each iteration of the algorithm. After the completion of the whole disk buffer, of size b, the variable *probedPages* is reset to zero. The pseudo-code for the proposed R-MESHJOIN is defined in Algorithm 2. Normally the algorithm runs for an infinite amount of time (line 3). For each iteration the algorithm examines the value of *probedPages* and if it is equal to zero or equal to b, reads new b disk pages from R and resets the value of *probedPages* to zero (lines 4-7). In addition to that, in each iteration the algorithm also scans w stream tuples from the stream buffer (line 8). Before loading these new scanned tuples of the stream S into the hash table H, the algorithm checks the status of the queue Q. If Q is already full the algorithm dequeues the addresses of the oldest w tuples from Q and removes the corresponding tuples from H (lines 9-12). Once the oldest tuples have been removed the algorithm appends the new scanned stream tuples into the hash table along with enqueuing their pointer addresses into the queue (lines 13-14). For each iteration the algorithm probes all the disk tuples of one logical partition of size b_P into the hash table and generates the output in the case of a match (lines 15-17). Finally, the algorithm increases the value of variable *probedPages* by b_P (line 18).

Algorithm 2 R-MESHJOIN

Input: A master data R and a Stream S
Output: $S \bowtie R$

Parameters: w tuples of S and b pages of R
Method:

```
1: probedPages ← 0
2: b_P ← Size of each logical partition of the disk buffer
3: while (true) do
4:    if probedPages mod b = 0 then
5:       READ b pages of R into the disk buffer
6:       probedPages ← 0
7:    end if
8:    READ w tuples of S into the stream buffer
9:    if Q is full then
10:      DEQUEUE w pointers from Q
11:      REMOVE relevant tuples from H
12:   end if
13:   ENQUEUE w pointers into Q
14:   ADD relevant tuples into H
15:   for each tuple r in b_P pages of R do
16:      Output r ⋈ H
17:   end for
18:   probedPages ← probedPages + b_P
19: end while
```

4.3.2 Understanding the Real Dependency

Both algorithms, MESHJOIN and R-MESHJOIN, distribute the total available memory among several distinct components. For each component, if seen in isolation, a larger amount of memory is preferable. If, however, a fixed total memory budget is assumed then a dependency is created for the distribution of memory between the different components, since more memory for one component means inevitably less memory for some other components. It is important to understand the exact nature of this dependency.

In MESHJOIN the number of partitions in the queue is equal to the total number of iterations required to bring the whole of R into memory, and that number of partitions in the queue depends inversely on the size of the disk buffer. The formulas of MESHJOIN are more involved because they express the queue size, not in bytes or number of tuples, but in the number of partitions in the queue, while these partitions change their size. Simply knowing the total number of partitions in the queue means that it is hard to find

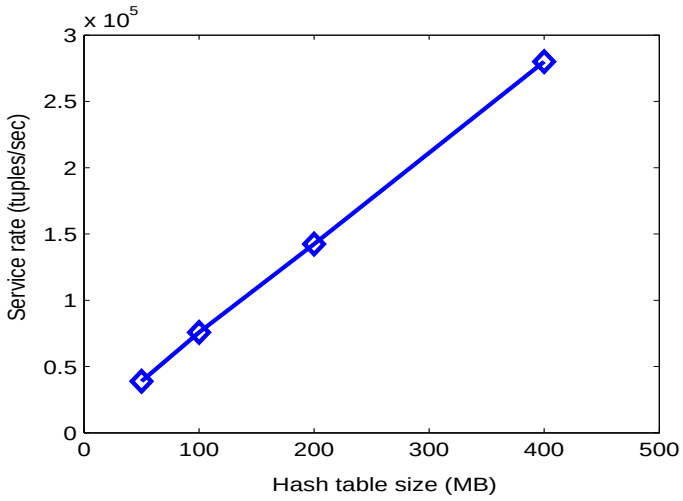


Figure 4.5: Effect of hash table size on join performance

out the size of the queue unless we know the value of w, that is, the total tuples in each partition. In Figure 4.1 the main components that directly affect the performance of the join are the hash table and the disk buffer. If the allocated memory for the hash table is increased, the throughput increases accordingly, as shown in Figure 4.5. The performance of the join directly depends on the memory size for the hash table.

The performance of the join is increased with an increase in the size of the disk buffer, as shown in Figure 4.6. In the experiment the memory limits for the hash table and the stream buffer remain fixed, while the total allocated memory for the join execution increases as the size of the disk buffer increases. From the figure, it can be seen that the performance improves up to a particular size of the disk buffer and further increments in its size do not improve the performance significantly. The plausible reason for this saturation is that the probing of H with disk pages is executed tuple by tuple. Therefore keeping more disk pages in the memory does not increase the performance once a reasonable buffer size is reached. On the other hand the buffer size cannot be reduced down to the tuple size because of the extra overhead of the I/O cost. Besides its slightly better performance, one of the main contributions of R-MESHJOIN is a better understanding of the true nature of this dependency. The main dependency, it turns out, is between the size of the buffer for R and the size of the hash table that is used for the join algorithm. In essence, the tuning task is to find the optimal distribution of the total memory between these two components. The important advancement in understanding is that neither the total size of R nor the stream buffer size plays an important role. This was made possible only

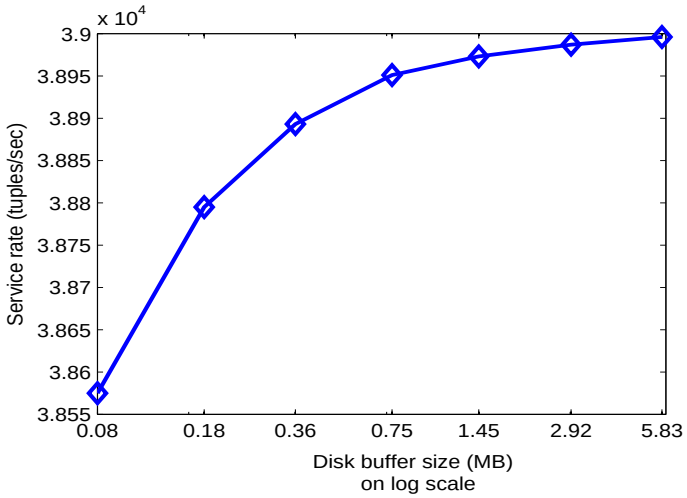


Figure 4.6: Effect of disk buffer size on join performance

through the juxtaposition of MESHJOIN and R-MESHJOIN, because in MESHJOIN these other parameters, notably the size of the table R, have a spurious influence on the optimisation process. This influence is spurious because it is owed solely to the lack of one degree of freedom in the MESHJOIN algorithm. The non-optimality of MESHJOIN and the slight improvement of R-MESHJOIN are closely related to this difference in degrees of freedom.

This simple picture is, however, complicated by an additional difficulty in turning this dependency into an optimal tuning of the algorithm. This additional difficulty is the fact that certain parameters must be in a whole-number ratio to other parameters. For instance, the disk buffer size for table R must be an integer fraction of the total size of R. Similarly, the cells in the queue must in turn be in a whole-number ratio to the aforementioned fraction. R-MESHJOIN introduces precisely more flexibility that is necessary to clearly separate the fundamental dependency between the disk buffer and the hash table on the one hand from the rather technical question of finding a sufficiently close whole-number relationship between the different parameters.

4.4 Cost Model and Tuning

This section presents the cost calculations for R-MESHJOIN in terms of memory and processing. The cost model presented here follows the style used for MESHJOIN [90, 91]. Once the cost has been calculated the algorithm is tuned in order to obtain the maximum

performance. Most of the symbols used here have already been mentioned in Chapter 3. However, some additional symbols are specified in Table 4.1.

4.4.1 Memory Cost

In R-MESHJOIN, the maximum portion of the total memory is used for the hash table H while a much smaller portion is used for the disk buffer and the queue. Once the available memory for the join has been determined, a specific part of the total memory is assigned to the disk buffer and then the rest of the memory is assigned among all the other components. The stream buffer component reserves a very small amount of the total memory (0.05 MB memory for the stream buffer is sufficient even up to 1GB, allocated for the whole join); therefore it is ignored for the moment. The other component is the queue, which is directly connected with the hash table and linearly depends on the hash table with respect to size. Therefore once the size for the disk buffer has been determined the memory reserved for the hash table and the queue can be calculated using the following relations.

Total allocated memory (bytes) = M

Disk buffer size (bytes) = $b \cdot v_P$

Memory for the hash table plus the queue (bytes) = $M - b \cdot v_P$

To distinguish the amount of memory used for the hash table and the queue, it is first necessary to know the ratio of both with respect to the amount of memory taken for one tuple. This ratio r can be specified simply by knowing the pointer size for each tuple in the queue and the stream tuple size v_S in the hash table, along with its fudge factor f, as given below. The fudge factor is an implementation overhead for the hash table.

$$r = \frac{\text{sizeof}(ptr)}{f \cdot v_S}$$

Hash memory (bytes) = $H_m = \alpha(M - b \cdot v_P)$

Queue memory (bytes)= $Q_m = (1 - \alpha)(M - b \cdot v_P)$

Number of partitions in the queue = $Q_p = k \cdot l$

Number of tuples in the hash table = $H_t = \frac{H_m}{v_S \cdot f}$

The total memory used by R-MESHJOIN can be calculated using Equation 4.1.

$$M(\text{bytes}) = b \cdot v_P + \alpha(M - b \cdot v_P) + (1 - \alpha)(M - b \cdot v_P) \tag{4.1}$$

Table 4.1: Some new symbols used in R-MESHJOIN

Parameter name	Symbol
Total number of tuples in R (millions)	R_t
Number of logical partitions in the disk buffer	l
Size of each logical partition in the disk buffer (pages)	b_P
Memory weight for hash table	α
Memory weight for queue	1-α
Memory reserved by hash table (bytes)	H_m
Memory reserved by queue (bytes)	Q_m
Number of partitions in the queue	Q_p
Total number of tuples in the hash table	h_S

Table 4.2: Memory measurements for three different cases of R-MESHJOIN

b *MB*	H *MB*	Q *MB*	w *MB*	*Total memory MB*
0.31	48.47	1.21	0.0005	49.9905
0.43	97.14	2.42	0.0013	99.9913
0.52	145.83	3.64	0.0024	149.9924

Currently we are not including the memory reserved for the stream buffer because it is small, however, it can be calculated if w is known. The value for w can be calculated using the values of H_t and Q_p using Equation 4.2.

$$w(\text{tuples}) = \frac{H_t}{Q_p} \tag{4.2}$$

In order to explain this further the memory cost has been calculated for three different cases. Table 4.2 depicts the concrete values for the major components and the total memory allocated for R-MESHJOIN.

4.4.2 Processing Cost

To calculate the processing cost for R-MESHJOIN the processing cost for one loop iteration is calculated first. As described in Section 4.3 the disk buffer is divided into l logical partitions and the disk input is taken after every l loop iterations of the algorithm. Therefore in principle the I/O cost to read b pages should be divided among l loop iterations. In order to calculate the cost for one loop iteration the major components are:

Cost to read b pages= $c_{I/O}(b)$

Cost to remove w tuples from H and Q= $w \cdot c_E$

Table 4.3: Processing cost of one loop iteration in three different cases of R-MESHJOIN

w *tuples*	$\frac{c_{I/O}(b)}{l}$ *nanosecs*	$w \cdot c_E$ *nanosecs*	$w \cdot c_S$ *nanosecs*	$w \cdot c_A$ *nanosecs*	$\frac{b}{l}(\frac{v_P}{v_R})c_H$ *nanosecs*	$\frac{b}{l}(\frac{v_P}{v_R})c_O$ *nanosecs*	c_{loop} *secs*
25	3318	11454	2514	12851	607116	5496	0.000642749
68	3674	25422	3632	26819	831048	7454	0.000898049
125	3731	53917	5029	54476	1018704	9136	0.001144993

Cost to read w tuples of the stream $S = w \cdot c_S$

Cost to append w tuples into Q and $H = w \cdot c_A$

Cost to probe one logical partition, each of size b_P, of the disk buffer with $H = \frac{b}{l}(\frac{v_P}{v_R})c_H$

Cost to generate the output for one logical partition of the disk buffer= $\frac{b}{l}(\frac{v_P}{v_R})c_O$

By aggregating all above costs, the total cost for one loop iteration can be calculated using Equation 4.3. In R-MESHJOIN all the processing costs are measured individually in nanoseconds and the total cost is converted into seconds as in Equation 4.3.

$$c_{loop}\ (\text{secs}) = 10^{-9}[\frac{c_{I/O}(b)}{l} + w(c_E + c_S + c_A) + \frac{b}{l}(\frac{v_P}{v_R})(c_H + c_O)] \tag{4.3}$$

In each iteration since the algorithm processes w stream tuples, the service rate μ can be calculated using Equation 4.4.

$$\mu = \frac{w}{c_{loop}} \tag{4.4}$$

In Table 4.3, the processing cost is measured for three different memory budgets set out in Table 4.2. On the basis of the given cost the service rate is also calculated using the formula described in Equation 4.4. If we consider the first row of the table using Equation 4.4 the service rate μ is: $\mu = \frac{25}{0.000642749} = 38895(\text{tuples/sec})$

4.4.3 Tuning of the Disk Buffer

In order to achieve maximum performance of the proposed R-MESHJOIN, the disk buffer is tuned using the proposed cost model while the total memory budget is fixed. Using Equations 4.2, 4.3 and 4.4 the service rate μ can be specified as a function of b, the size of the disk buffer, as in Equation 4.5. Therefore by assigning different values to the parameter b it is possible to find the required value for the disk buffer size on which the

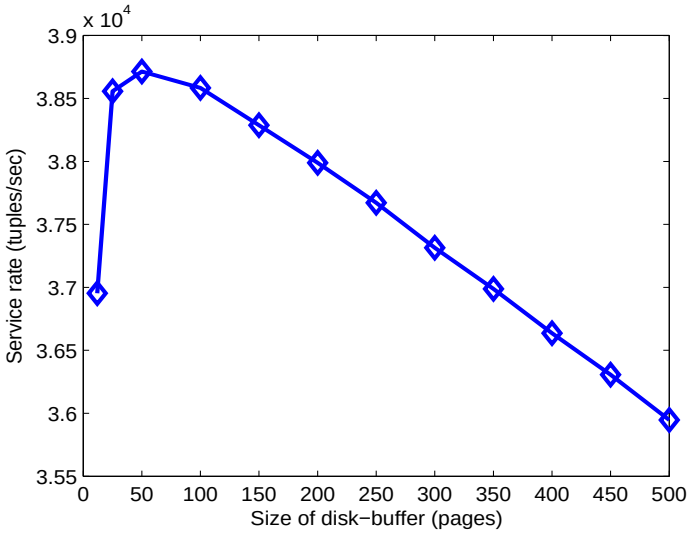


Figure 4.7: Disk buffer tuning within fixed memory budget

function returns the maximum service rate.

$$\mu = \frac{\alpha(M - b \cdot v_P)}{k \cdot l \cdot v_S \cdot f \cdot c_{loop}} \tag{4.5}$$

An experimental analysis has also been conducted to measure the performance depending on the size of the disk buffer within a fixed memory budget. Figure 4.7 demonstrates the performance of R-MESHJOIN while increasing the size of the disk buffer sequentially. It can be observed that an increment in the size of the disk buffer increases the service rate up to a certain value and for further increments in the size of the disk buffer, the service rate decreases. This performance behavior can be explained as follows. For a small disk buffer size the efficiency of reading the disk pages is low, but due to the memory constraints w cannot be increased to balance this increasing I/O cost. In the case where the disk buffer gets larger it reduces the memory available for the hash table in order to accommodate the data within the fixed-size memory budget.

4.5 Experiments

To validate the argument, described in this chapter, an extensive experimental evaluation of R-MESHJOIN has been carried out, using synthetic data sets. This section describes the methodology, used to implement the algorithm, and analyses the results.

4.5.1 Experimental Setup

The hardware used in our experiments is the same as described in Chapter 3. However, the data specifications and the results measurement strategy are slightly different. Details of both are given below.

Data specification: The performance of R-MESHJOIN has been analysed using synthetic data. The master data R is stored on disk in a text file format while the stream data is generated at run time using the same random-number generating script as that which is used in Chapter 3. In these experiments the size of R is considered to be from 0.5 million tuples to 16 million tuples while the size for each tuple is 120 bytes. The size of each tuple in the stream is considered to be 20 bytes. While each pointer size in the queue is 4 bytes and the value of the fudge factor is 8. Currently one assumed characteristic related to the stream S is that there will be no intermittent arrival of stream data during join execution. In the experiments the tuning module executes first, before starting the execution of R-MESHJOIN. The tuning module basically tunes the disk buffer to an optimal value. Once the optimal size for the disk buffer has been determined then the memory is divided among all the join components using the proposed memory distribution strategy, described in Section 4.4.

Measurement strategy: The performance or service rate of the join is measured by calculating the number of tuples processed in a unit second. In each experiment the algorithm completes two rounds of R and the measurements are taken from the start of the second round. For greater accuracy the confidence interval is calculated by considering a 95% accuracy rate. Moreover, during the execution of the algorithm no other application is assumed to run in parallel.

4.5.2 Experimental Results

Cost validation: In this experiment the cost model has been validated by comparing the calculated cost with the measured cost. Figure 4.8(a) presents the comparison of these costs. In the figure it is demonstrated that the calculated cost closely resembles the measured cost.

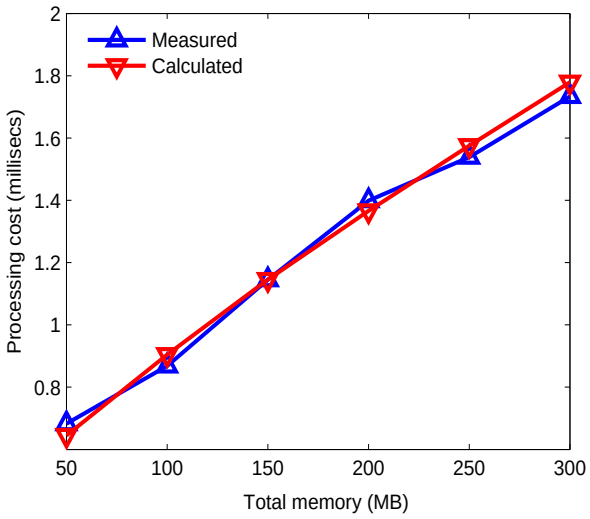
Comparison of optimal disk buffer sizes for different memory budgets: In this part of our experiment the disk buffer has been tuned for a series of memory budgets

using both approaches and the optimal values for the disk buffer sizes have been measured accordingly. Figure 4.8(b) demonstrates the comparisons between the optimal values of the disk buffer sizes using the MESHJOIN and R-MESHJOIN algorithms. From the figure it can be observed that for small memory budgets the difference between the optimal values of both approaches is small (the optimal value in the case of R-MESHJOIN is less than that of MESHJOIN). But as the memory budget increases, this difference also increases. The reason for this is that, in the case of MESHJOIN, it is difficult to determine the accurate optimal value for the disk buffer size, due to the influences of irrelevant factors in the tuning phase.

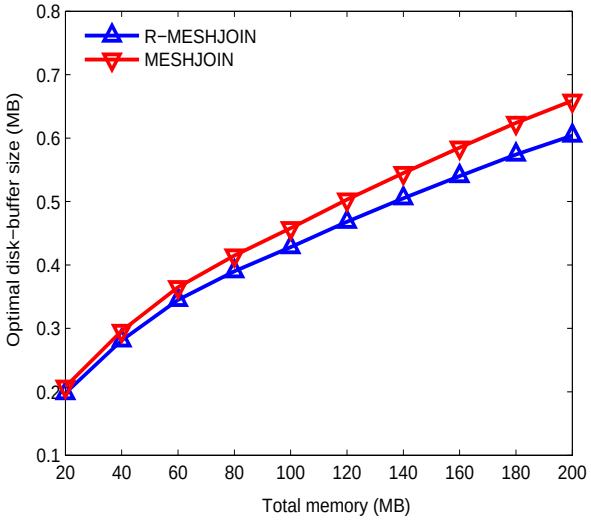
Performance comparison with MESHJOIN using different memory budgets: In this experiment the performance of R-MESHJOIN has been compared with MESHJOIN using different memory budgets. Figure 4.8(c) depicts the comparison between the two approaches. For small memory budgets the improvement in performance using R-MESHJOIN is not noticeable but as the total memory budget gets larger this difference in performance becomes more visible.

Performance comparison with MESHJOIN using different sizes of R: In this experiment the performance of both algorithms has been compared for the different sizes of master data R. From Figure 4.8(d), for a small size of R the performance improvement in case of R-MESHJOIN can be observed clearly. For the larger sizes of R the performance of R-MESHJOIN is better still. However, this may not be apparent on the graph because of the scale of the graph. In addition to the comparison analysis, the overall performance in both approaches has been analysed with respect to the allocated memory and the size of the master data. The experimental results shown in Figure 4.8(c)and 4.8(d) also verify the theoretical results described in Section 4.3.

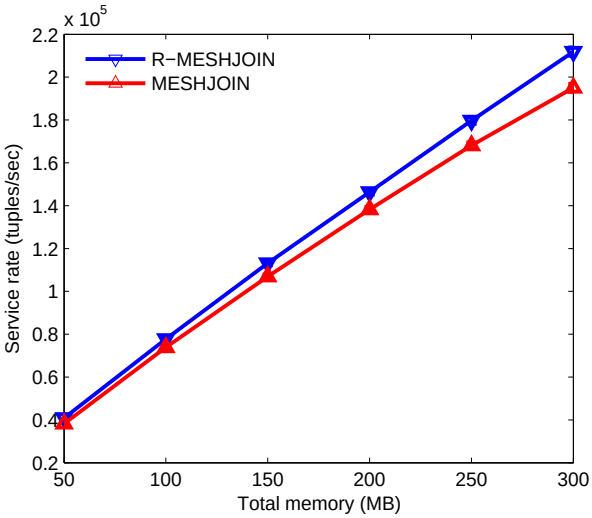
Disk buffer analysis: To further strengthen the argument an analysis of disk buffer size has been presented against the different sizes of R. Table 4.4 shows the results of this analysis (for simplicity it is also shown by graph in Figure 4.9). From the table (and figure) it is clear that the size of the disk buffer $b \cdot v_P$ does not change with a change in the size of R, which is consistent with expectation. Moreover, by introducing a new component l the variation in the size of R affects neither the total number of partitions Q_p nor the size of each partition w in the queue.



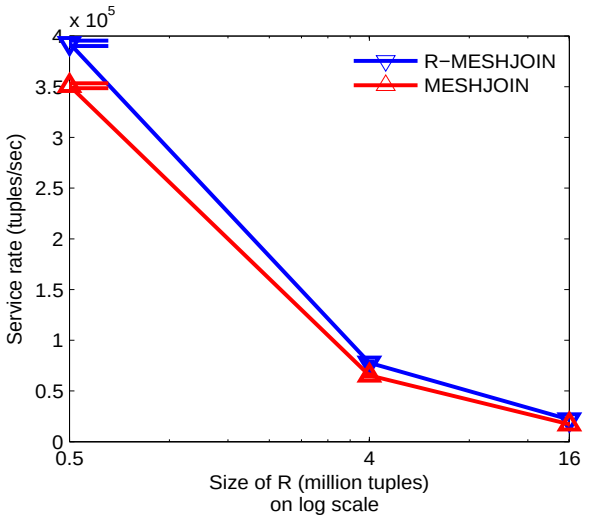
(a) R-MESHJOIN: Measured Vs calculated cost



(b) Comparison between the optimal values for the disk buffer size



(c) Performance comparison with 95% confidence interval in the case of different memory budgets



(d) Performance comparison with 95% confidence interval in the case of different sizes of R

Figure 4.8: Experimental results

Table 4.4: Disk buffer analysis for different sizes of R

M MB	R_t Millions	$b \cdot v_P$ MB	H_m MB	Q_m MB	k	l	Q_p	w
100	0.5	0.427	97.145	2.423	134	64	8576	75
100	1	0.427	97.145	2.423	268	32	8576	75
100	2	0.427	97.145	2.423	536	16	8576	75
100	4	0.427	97.145	2.423	1072	8	8576	75
100	8	0.427	97.145	2.423	2144	4	8576	75
100	16	0.427	97.145	2.423	4288	2	8576	75

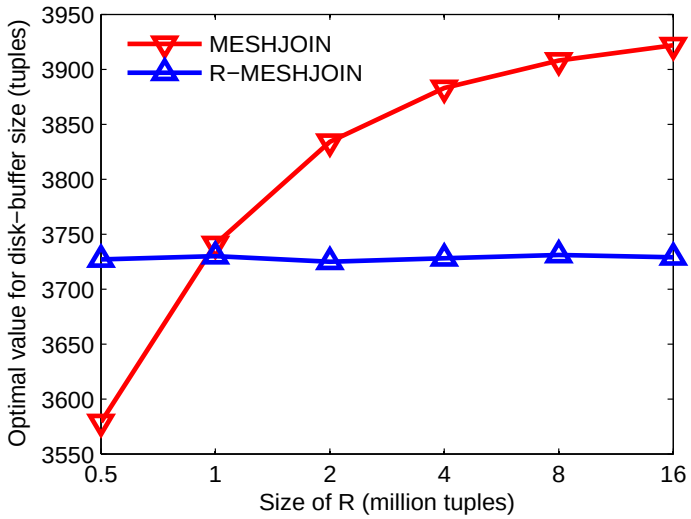


Figure 4.9: Disk buffer analysis for different sizes of *R*: MESHJOIN vs R-MESHJOIN

4.6 Summary

This chapter presents an analysis of the MESHJOIN algorithm, focusing on the issue of dependencies between its components. In the process of clarifying the working of each component, a modification called R-MESHJOIN has been proposed, which helps to analyse MESHJOIN both theoretically and experimentally. The fact that R-MESHJOIN delivers a certain small, but both theoretically and empirically unambiguous improvement is an important corroboration of our analysis. First and foremost, however, the nature of the dependency between the different components in all MESHJOIN-type algorithms is now better understood. The cost model for the proposed R-MESHJOIN has been presented and the tuning of a key component called the disk buffer has also been performed based on the cost model. Experiments have proved that by implementing the true dependency between the components the performance of the algorithm has also been improved. Finally, the cost model has been validated by comparing it with empirical costs.

5

A New HYBRIDJOIN

5.1 Introduction

The MESHJOIN [90, 91] algorithm is, in principle, a hash join, where the stream serves as the build input and the master data serves as the probe input. The main contribution of this algorithm is a staggered execution of the hash table build and an optimisation of the disk buffer for the master data.

5.1.1 Disk Access Strategy in MESHJOIN

MESHJOIN was proposed for joining a stream with a slowly changing table using limited main memory. This algorithm is an interesting candidate for a resource-aware system setup. The MESHJOIN algorithm also has few requirements with respect to the organisation of the master data table. The algorithm makes no assumptions about data distribution or the organisation of the master data. Experiments by the MESHJOIN

authors have shown that the algorithm performs less efficiently with non-uniform data. However, the problem that is being addressed here is that the MESHJOIN performance is directly coupled to the size of the master data table, and is inversely proportional to the size of the master data table. This is an undesired behavior if the master data becomes very large, and the analysis will show that it is indeed an unnecessary behavior. The problem becomes even more obvious if there is a large portion of the master data table that is never joined with the stream data. In MESHJOIN, if one contiguous half of the master data is unused, its presence halves the performance. This situation can easily arise if the master data table is storing data for long-term availability, and only a fraction is used in the current business process. A typical scenario would be catalogue data for seasonal products. This reduction of performance is undesirable, especially from a resource-consumption viewpoint, since the algorithm uses the same resources for only half the performance. This would put the burden on the administrator to clean the master data meticulously in order to optimise system performance. Therefore it would be a great advantage to have an algorithm that shares the advantages of MESHJOIN, but adapts itself to certain situations, for example if the algorithm could be made more sensitive to the usage of the master data.

To determine the access rate of disk pages of R an experiment has been performed using a benchmark that is based on Zipfian distribution. The detail of this benchmark is provided in Section 5.3. In this experiment it has been assumed that R is sorted in ascending order with respect to the frequency of join attribute values in the stream. The rate of use has been measured for the same size of partitions (each partition contains *20* pages) at different locations of R. From the results shown in Figure 5.1 it can be observed that the rate of page-use decreases towards the end of R. The MESHJOIN algorithm does not consider this factor and loads the unused or less-used pages of R into memory with equal frequency, which increases the *processing time* for every stream tuple in the queue due to extra disk *I/O*. *Processing time* is the time that every stream tuple spends in the join window from loading to join, without including any delay due to the low arrival rate of the stream. The average *processing time* in the case of MESHJOIN can be estimated using the following formula.

Average *processing time (secs)* $= \frac{1}{2}(seektime + accesstime)$ for the whole of R

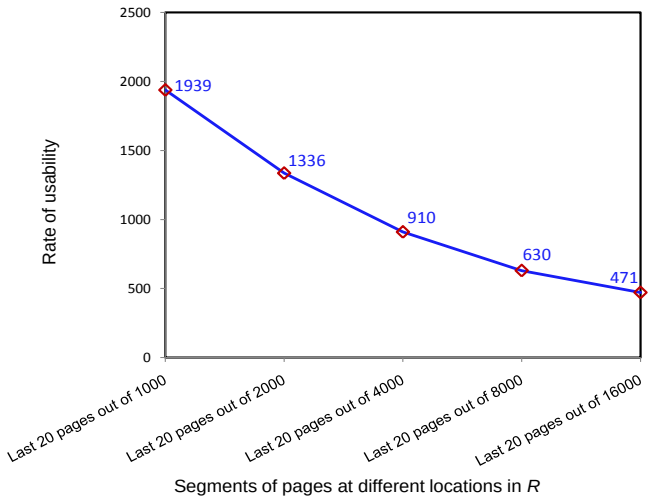


Figure 5.1: Measured rate of page use at different locations of *R* while the size of total *R* is *16000* pages

5.1.2 Intermittency in MESHJOIN

In addition to the above issue, MESHJOIN cannot deal with bursty input streams effectively. There are many practical scenarios where the stream arrival rate is bursty and the distribution of data items in the stream is non-uniform [32, 63, 65, 67, 74, 89, 98, 106, 115, 116, 117]. In MESHJOIN a disk invocation occurs when the number of tuples in the stream buffer is equal to or greater than the stream input size w. If the input stream has an intermittent or low arrival rate (λ), the tuples already in the queue need to wait longer due to a disk invocation delay. This *waiting time* affects the performance negatively. The average *waiting time* can be calculated using the following formula.

Average *waiting time (secs)* $= \frac{w}{\lambda}$

The Index Nested Loop Join (INLJ) [92] algorithm traditionally is used for non-stream data, but it can easily be set up so that it joins a continuous data stream with a master data, which is capable of dealing with intermittent data streams. However, every index needs to be considered as non-clustered with respect to the stream data, because stream data arrive in the order that the updates are performed. This is a natural assumption, for example, purchases are not sorted on product numbers. INLJ is known to be inefficient for non-clustered index access. The disk I/O cost cannot be amortised over multiple tuples of the stream and eventually produces a low service rate.

Based on these observations a new stream-based join algorithm, called HYBRIDJOIN

(Hybrid Join) [83][1] has been proposed in this chapter. The key difference between HYBRIDJOIN and MESHJOIN is that HYBRIDJOIN does not read the entire master data sequentially but instead accesses it using an index. This can reduce the disk *I/O* cost by guaranteeing that every partition read from the master data is used to remove at least one stream tuple from the memory, while in MESHJOIN there is no such guarantee. Further, performance in HYBRIDJOIN is not affected if a large set of unused data is added to the master data table. To amortise the disk read over many stream tuples, the algorithm performs the join of disk partitions with all the stream tuples currently in memory. This approach guarantees that HYBRIDJOIN is never asymptotically slower than MESHJOIN. HYBRIDJOIN can perform worse than MESHJOIN in only one case when the stream data is completely uniform. Moreover HYBRIDJOIN can provide the worst case constant, which can be determined by disk seek time. In addition, in HYBRIDJOIN, unlike MESHJOIN, the disk load is not synchronised with the stream input, providing better service rates for bursty streams.

One of the concerns in this chapter is to understand the relative performance of the MESHJOIN and HYBRIDJOIN algorithms. The main result of the analysis is that HYBRIDJOIN performs better on a non-uniform data set that models a Zipfian distribution. As noted above HYBRIDJOIN is unaffected by contiguous unused master data. Therefore it is easy to create data sets where HYBRIDJOIN is arbitrarily better than MESHJOIN. However, in order to test the HYBRIDJOIN algorithm in a scenario that is not biased against any algorithm, it is desirable to look for characteristics of data that are considered ubiquitous in real world scenarios. A Zipfian distribution of the foreign keys in the stream data matches with distributions that are observed in a wide range of applications [3]. A data generator has therefore been created which can produce such a Zipfian distribution. A Zipfian distribution is parameterized by the exponent of the underlying power law. Different exponents are observed in different scenarios and determine whether the distribution is considered to have a short tail or a long tail. HYBRIDJOIN would give better performance from the outset for distributions with a short tail, therefore it was decided not to use a distribution with a short tail in order to not bias the experiment towards HYBRIDJOIN. Instead the most natural exponent observed in a variety of areas was chosen, including the original Zipf's Law in linguistics [66] that gave rise to the popular

[1] This work has been published in International Journal of Data Warehousing and Mining (IJDWM'11).

name of these distributions.

HYBRIDJOIN performs at most by a constant factor worse than MESHJOIN. The worst performance is obtained when the key distribution is completely uniform and an adaptive approach is not needed. MESHJOIN performs better in this case since it eliminates seek times by reading the master data sequentially.

The rest of the chapter is structured as follows. Section 5.2 presents the architecture, algorithm, theoretical analysis, cost model, and tuning of the proposed HYBRIDJOIN. The design and implementation of a benchmark for testing HYBRIDJOIN are described in Section 5.3. The experimental study is discussed in Section 5.4 and finally Section 5.5 presents a summary of the chapter.

5.2 HYBRIDJOIN

In Section 5.1 certain observations about the MESHJOIN and INLJ algorithms are identified. As a solution to the stated observations a new stream-based join algorithm called HYBRIDJOIN is proposed. HYBRIDJOIN achieves two major aims: (a) efficient strategy to access the master data R by loading only the useful part of R into memory, and (b) dealing with bursty streams effectively.

HYBRIDJOIN joins a master data R with a stream S. A sorted index by access frequency is assumed for the join attribute in R, and it is also assumed that the join attribute is unique within the master data. This is a very natural set of assumptions and matches the application domain described above, for example in key exchange applications. Requiring only sorting of index keeps the algorithm's assumptions small.

This section describes the architecture, pseudo-code and run-time analysis of the proposed algorithm. This section also presents the cost model that is used for estimating the cost for the algorithm, and for tuning the algorithm.

5.2.1 Memory Architecture

The memory architecture for HYBRIDJOIN is shown in Figure 5.2. The key components of HYBRIDJOIN are the disk buffer, the hash table, the queue and the stream buffer. The master data R and stream S are the inputs. This algorithm assumes that R is sorted

and has an index on the join attribute. The stream is used as the build input. This means that the algorithm keeps stream tuples in a hash table which occupies the largest share of the memory, and the hash table is filled with the next pending stream tuples up to its full capacity. Additionally the algorithm keeps identifiers of the stream tuples in a queue which allows random deletion. The simplest implementation is a doubly-linked-list. The role of the stream buffer is simply to hold the fast stream if necessary.

HYBRIDJOIN is an iterative algorithm, and in each iteration it uses a partition of the master data R as a probe input. For that purpose, the partition is loaded into the disk buffer. After that, the algorithm performs the typical operation of a hash join, i.e., it loops over all the tuples of the disk buffer and looks them up in the hash table. In the case of a match, the algorithm generates the resulting stream tuple as an output. Also, in each iteration, HYBRIDJOIN evicts stream tuples that have been matched. This is justified through the assumption that the join attribute is unique in R. Evicting a tuple means it is deleted from the hash table and the queue. The algorithm also keeps a counter w of the evicted tuples. After processing the whole disk buffer, the algorithm reads w new tuples from the stream buffer and loads them into the hash table, along with entering their identifiers in the queue.

When choosing the next partition of R, HYBRIDJOIN looks at the join attribute of the oldest stream tuple in the queue. Using the index, it loads the partition of R with that join attribute value into the disk buffer. It is this last step which makes HYBRIDJOIN adaptive, because in HYBRIDJOIN every loaded partition removes at least one stream tuple from the memory. As a simple example, assume that R has a section that is not referred to in the stream, for example an obsolete group of products. In MESHJOIN, this section would still be loaded, while in HYBRIDJOIN it would not be loaded, because no stream tuple will trigger the loading of that section. Also when the disk partition is loaded into memory using the join attribute value from the queue as an index, instead of only matching one tuple as in INLJ, the algorithm checks the disk partition against all the stream tuples in the queue. This helps to amortise the fast arrival stream.

To deal with the intermittencies in the stream, for each iteration the algorithm loads a disk partition into memory and checks the status of the stream buffer. In the case where no stream tuples are available in the stream buffer the algorithm will not stop but continues its working until the hash table becomes empty. However, the queue keeps on

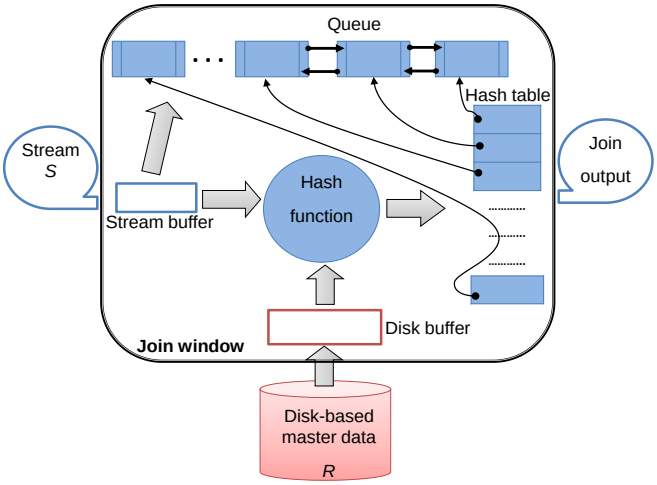


Figure 5.2: Memory architecture for HYBRIDJOIN

shrinking continuously and will become empty when all tuples in the hash table have been joined. When the stream resumes, the queue begins to grow again. However, in MESHJOIN every disk input is dependent on stream input.

HYBRIDJOIN works for any data distribution, as MESHJOIN does. However, in practice, certain distributions are more common. Current research has shown that sales data typically follows a power law, or Zipfian distribution [3]. The power law is characterized by its exponent. For an exponent <1 the distribution is said to have a long tail, for an exponent >1 the distribution has a short tail. For exponent 1 we get the distribution of Zipf's law, which gave rise to the general term Zipfian distribution. In sales, the 80/20 rule is used to model the scenario where the frequency of selling a small number of products is significantly higher compared to the rest of the products, often simplified in the 80/20 rule. The 80/20 rule corresponds to an exponent slightly smaller than 1 [66].

The aim here is to describe an algorithm that takes advantage of the likely distribution of the data. Therefore a dataset generator was created that can create artificial data sets, following a power law with an exponent that can be chosen freely. In all the experiments reported in this chapter, the master data is assumed to be sorted with respect to the access frequency.

5.2.2 Algorithm

Once the memory has been distributed among the join components HYBRIDJOIN starts its execution, according to the procedure defined in Algorithm 3. Initially since the hash

table is empty, h_S is assigned to stream input size w where h_S is the total number of slots in the hash table H (line 1). The algorithm consists of two loops: one is called the outer loop while the other is called the inner loop. The outer loop, which is an endless loop, is used to build the stream input in the hash table (line 2), while the inner loop is used to probe the disk tuples in the hash table (line 9). In each outer loop iteration, the algorithm examines the availability of stream input in the stream buffer. If stream input is available, the algorithm reads w tuples of the stream and loads them into the hash table while also placing their join attribute values in the queue. Once the stream input is read the algorithm resets the value of w to zero (lines 3-6). The algorithm then reads the oldest value of a join attribute from the queue and loads a disk partition into the disk buffer, using that join attribute value as an index (lines 7,8). After the disk partition has been loaded into memory the inner loop starts and for each iteration of the inner loop the algorithm reads one disk tuple from the disk buffer and probes it into the hash table. In the case of a match, the algorithm generates the join output. Since the hash table is a multi-hash-map, there may be more than one match against one disk tuple. After generating the join output the algorithm deletes all matched tuples from the hash table, along with the corresponding nodes from the queue. Finally, the algorithm increases w with the number of vacated slots in the hash table (lines 9-15).

5.2.3 Asymptotic Runtime Analysis

This section presents the asymptotic runtime comparison of HYBRIDJOIN with that of MESHJOIN and INLJ. The time needed to process a stream segment is used as a unit of measurement. The time needed to process a single tuple is the inverse of the service rate, which is the number of tuples processed in a time interval. The unit of measurement used here has the advantage that "smaller is better", in accordance with common usage in the asymptotic analysis of algorithms. Consider a concrete stream prefix s. The time needed to process a stream section s is denoted as *MEJ(s)* for MESHJOIN, as *INLJ(s)* for index nested loop join, and as *HYJ(s)* for HYBRIDJOIN. Every stream section represents a binary sequence, and by viewing this binary sequence as a natural number, asymptotic complexity classes can be applied to the functions above. Note therefore that the following theorems do not use functions on input lengths, but on concrete inputs. The resulting

Algorithm 3 HYBRIDJOIN

Input: A master data R with an index on join attribute and a stream of updates S
Output: $S \bowtie R$
Parameters: w tuples of S and a partition of R
Method:

```
1: w ← h_S
2: while (true) do
3:    if (stream available) then
4:       READ w tuples form the stream buffer and load them into H while enqueuing
         their join attribute values in Q.
5:       w ← 0
6:    end if
7:    READ the oldest join attribute value from Q.
8:    READ a partition of R into disk buffer using that join attribute value as an index.
9:    for each tuple r in the chosen partition do
10:      if r ∈ H then
11:         OUTPUT r ⋈ H
12:         DELETE all matched tuples from H along with the corresponding nodes from
            Q.
13:         w ← w+ number of matching tuples found in H
14:      end if
15:   end for
16: end while
```

theorems imply analogous asymptotic behavior on input length, but are stronger than statements on input length. It is assumed that the setup for HYBRIDJOIN and for MESHJOIN is such that they have the same number h_S of stream tuples in the hash table, and accordingly in the queue.

Comparison with MESHJOIN:

Theorem 1: *HYJ(s) = O(MEJ(s))*

Proof: To prove the theorem, it has to be proved that HYBRIDJOIN performs no worse than MESHJOIN. The cost of MESHJOIN is dominated by the number of accesses to R. For analysing asymptotic run-time, random access of disk partitions is as fast as sequential access (seek time is only a constant factor). For MESHJOIN with its cyclic access pattern for R, every partition of R is accessed exactly once after every h_S stream tuples. It is necessary to show that for HYBRIDJOIN no partition is accessed more frequently. For that an arbitrary partition p of R at the time it is accessed by HYBRIDJOIN is considered. The stream tuple at the front of the queue has some position i in the stream. There are h_S stream tuples currently in the hash table, and the first tuple of the stream that has

not yet been read into the hash table has position $i+h_S$ in the stream. All stream tuples in the hash table are joined against the master data tuples on p, and all matching tuples are removed from the queue. Now the earliest time that p could be loaded again by HYBRIDJOIN has to be determined. For p to be loaded again, a stream tuple must be at the front of the queue, and has to match a master data tuple on p. The first stream tuple that can do so is the previously mentioned stream tuple with position $i+h_S$, because all earlier stream tuples that match data on p have been deleted from the queue. This proves the theorem.

Comparison with INLJ:

Theorem 2: *HYJ(s) = O(INLJ(s))*

Proof: INLJ performs a constant number of disk accesses per stream tuple. For the theorem it suffices to prove that HYBRIDJOIN performs not more than a constant number of disk accesses per stream tuple as well. The first to be considered here are those stream tuples that remain in the queue until they reach the front of the queue. For each of these tuples, HYBRIDJOIN loads a part of R and hence makes a constant number of disk accesses. For all other stream tuples, no separate disk access is made. This proves the theorem.

The theorems show that, except for a single constant factor c, HYBRIDJOIN performs on each individual input at least as well as either of the other two algorithms. The maximum factor is determined by the ratio of continuous disk access time to random disk access time for different disk portions. This is a free parameter in the cost model. In practice it depends on the technical parameters of the disk used, particularly the seek time, and on the choice of the disk portions that are loaded in one step. In the setup used here the factor is smaller than 2 for Theorem 1 and smaller than 5 for Theorem 2, i.e. even in the worst case, HYBRIDJOIN would be at most 2 times slower than MESHJOIN and at most 5 times slower than index nested loop join.

5.2.4 Cost Model

This section explains the general formulas used to calculate the cost for HYBRIDJOIN. Since it is important to compare this cost model with the cost model presented for MESHJOIN in [90, 91], the same notations are used here where possible and the cost

is also calculated in terms of memory and processing time. Equation 5.1 describes the total memory used to implement the algorithm (except the stream buffer). Equation 5.3 calculates the processing cost for w tuples, while the average size for w can be calculated using Equation 5.2. Once the processing cost for w tuples has been measured, the service rate μ can be calculated using Equation 5.4. The symbols used to measure the cost have already been specified in Chapter 3 and Chapter 4.

Memory Cost

In HYBRIDJOIN, the largest portion of the total memory is used for the hash table H while a comparatively smaller amount is used for the disk buffer. The queue size is linear in the hash table size, but considerably smaller. The separate size for each of them can be calculated easily.
Memory reserved for the disk buffer *(bytes)*= v_P
Memory reserved for the hash *(bytes)*= $\alpha(M - v_P)$
Memory reserved for the queue *(bytes)*= $(1 - \alpha)(M - v_P)$
The total memory used by HYBRIDJOIN can be determined by aggregating all the above.

$$M = v_P + \alpha(M - v_P) + (1 - \alpha)(M - v_P) \tag{5.1}$$

The memory reserved by the stream buffer is not included here because it is negligible (*0.05 MB* was sufficient in all these experiments). In order to explain this further the memory cost has been calculated for each component when the total available memory is *50MB*. Table 5.1 depicts the concrete values of memory required for the major components.

Processing Cost

This section presents the calculation of the processing cost for HYBRIDJOIN. To achieve this it is necessary to calculate the average stream input size w first.

Calculate average stream input size w: In HYBRIDJOIN the average stream input size w depends on the following four parameters.

- Size of the hash table, h_S (in tuples)
- Size of the disk buffer, d (in tuples)

Table 5.1: Memory measurements for major components of HYBRIDJOIN

Disk buffer *MB*	Hash table *MB*	Queue *MB*	M_{HYBRID} *MB*
0.059 (or 60KB)	47.66	2.24	49.959

- Size of the master data, R_t (in tuples)

- The exponent value for the benchmark, e

In our experiments w is directly proportional to h_S and d (where $d = \frac{v_P}{v_R}$), and is inversely proportional to R_t. Further details about these relationships can be found in Appendix A. The fourth parameter represents the exponent value for the stream data distribution as explained in Section 5.3, and using an exponent value equal to 1 the 80/20 Rule [3] can be formulated approximately for market sales. Therefore, the formula for w is:

$$w \propto \frac{h_S \cdot d}{R_t}$$

$$w = k\frac{h_S \cdot d}{R_t} \tag{5.2}$$

where k is a constant influenced by system parameters. The value of k has been obtained from measurements. In this setup it is 1.36.

On the basis of w the processing cost can be calculated for one loop iteration. In order to calculate the cost for one loop iteration the major components are:
Cost to read one disk partition= $c_{I/O}(v_P)$
Cost to probe one disk partition into the hash table= $\frac{v_P}{v_R}c_H$
Cost to generate the output for w matching tuples= $w \cdot c_O$
Cost to delete w tuples from the hash table and the queue= $w \cdot c_E$
Cost to read w tuples from the stream S= $w \cdot c_S$
Cost to append w tuples into the hash table and the queue= $w \cdot c_A$
By aggregation, the total cost for one loop iteration is:

$$c_{loop} = 10^{-9}[c_{I/O}(v_P) + \frac{v_P}{v_R}c_H + w \cdot c_O + w \cdot c_E + w \cdot c_S + w \cdot c_A] \tag{5.3}$$

Since the algorithm processes w tuples of the stream S in c_{loop} seconds, the service rate

Table 5.2: Processing cost for different operations of HYBRIDJOIN algorithm

w *tuples*	$c_{I/O}(v_P)$ *nanosecs*	$w \cdot c_E$ *nanosecs*	$w \cdot c_S$ *nanosecs*	$w \cdot c_A$ *nanosecs*	$\frac{v_P}{v_R} c_H$ *nanosecs*	$w \cdot c_O$ *nanosecs*	c_{loop} *secs*
109	9939229	572032	408641	391092	1308500	218000	0.012837

can be calculated by dividing w by the cost for one loop iteration.

$$\mu = \frac{w}{c_{loop}} \tag{5.4}$$

Table 5.2 provides measurements of the processing cost for different operations of the algorithm. Based on the processing cost the service rate has also been calculated using the formula described in Equation 5.4. By using Equation 5.4 the service rate μ is:

$$\mu = \frac{109}{0.012837} = 8491 \ tuples/sec$$

5.2.5 Tuning

Tuning of the join components is important in order to make efficient use of the available resources. In HYBRIDJOIN the disk buffer is the key component for tuning to amortise the disk I/O cost on fast input data streams. From Equation 5.4 the service rate depends on w and the cost c_{loop} required to process these w tuples. In HYBRIDJOIN, for a particular setting ($M = 50MB$) assuming that the size of R and the exponent value for the distribution are fixed ($R_t = 2Millions$ and $e = 1$), according to Equation 5.2 w then depends on the size of the hash table and the size of the disk buffer. Furthermore, the size of the hash table is also dependent on the size of the disk buffer, as shown in Equation 5.1. Therefore, using Equations 5.2, 5.3 and 5.4, the service rate μ can be specified as a function of v_P and the value for v_P at which the service rate is maximum can be determined by applying standard calculus rules.

Figure 5.3 shows the relationship between the I/O cost and service rate as measured in the experiments. It can be observed that in the beginning, for a small disk buffer size, the service rate is also small because there are fewer matching tuples in the queue. However, the service rate increases with an increase in the size of the disk buffer due to there being more matching tuples in the queue. After a particular value of the disk buffer

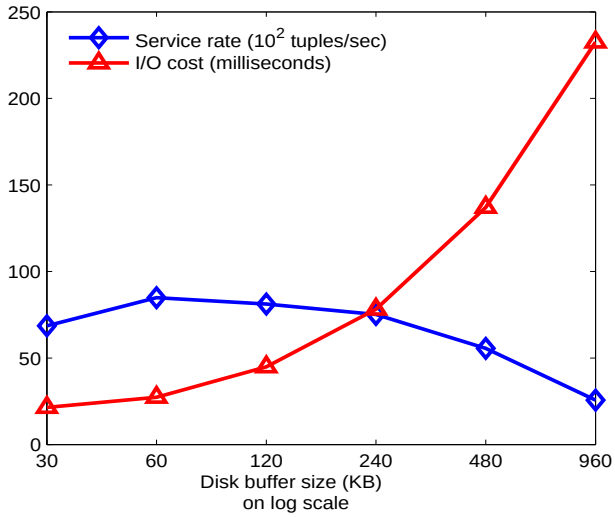


Figure 5.3: Tuning of the disk buffer

size is reached the trend changes and performance decreases with further increments in the size of the disk buffer. The plausible reason behind this decrease in performance is the rapid increase in the disk *I/O* cost and the decrease in memory size for the hash table.

5.3 Tests with Locality of Disk Access

A crucial factor for the performance of HYBRIDJOIN is the distribution of master data foreign keys in the stream. If the distribution is uniform, then HYBRIDJOIN may perform worse than MESHJOIN, but by a constant factor, in line with the theoretical analysis. Note however, that HYBRIDJOIN still has the advantage of being efficient for intermittent streams, while the original MESHJOIN would pause in intermittent streams, and leave tuples unprocessed for an open-ended period.

It is also obvious that HYBRIDJOIN has advantages if R contains unused data, for example, if there are old product records that are currently accessed very rarely, that are clustered in R. HYBRIDJOIN would not access these areas of R, while MESHJOIN accesses the whole of R.

More interesting, however, is whether HYBRIDJOIN can also benefit from more general locality. Therefore the question arises whether we can demonstrate a natural distribution where HYBRIDJOIN measurably improves over a uniform distribution, because of locality.

The popular types of distributions are Zipfian distributions, which exhibit a power law

similar to Zipf's law. Zipfian distributions are discussed as at least plausible models for sales [3], where some products are sold frequently while most are sold rarely. This kind of distribution can be modelled using Zipf's law.

A generator for synthetic data has been designed that models a Zipfian distribution, and it has been used to demonstrate that HYBRIDJOIN performance increases through locality, and that HYBRIDJOIN outperforms MESHJOIN.

In order to simplify the model, it has been assumed that the product keys are sorted in the master data table according to their frequency in the stream. This is a simplifying assumption that would not automatically hold in typical warehouse catalogues, but it provides a plausible locality behavior and makes the degree of locality very transparent.

Finally, in order to demonstrate the behavior of the algorithm under intermittence, a stream generator has been implemented that produces stream tuples with a timing that is self-similar.

This bursty generation of tuples models a flow of sales transactions which depends upon fluctuations over several time periods, such as market hours, weekly rhythms and seasons. The pseudo-code for the generation of the benchmark used here is shown in Figure 5.4. In the figure $S_{TREAM}G_{ENERATOR}$ is the main procedure while $G_{ET}D_{ISTRIBUTION}V_{ALUE}$ and $S_{WAP}S_{TATUS}$ are the sub-procedures that are called from the main procedure. According to the main procedure a number of virtual stream objects (in this case *10*), each representing the same distribution value obtained from the $G_{ET}D_{ISTRIBUTION}V_{ALUE}$ procedure, are inserted into a priority queue, which always keeps sorting these objects into ascending order (lines 5 to 7). Once all the virtual stream objects have been inserted into the priority queue the top most stream object is taken out (line 8). A loop is executed to generate an infinite stream (lines 9 to 18). In each iteration of the loop, the algorithm waits for a while (which depends upon the value of variable *oneStep*) and then checks whether the current time is more than the time when that particular object was inserted. If the condition is true the algorithm dequeues the next object from the priority queue and calls the $S_{WAP}S_{TATUS}$ procedure (lines 11 to 14). The $S_{WAP}S_{TATUS}$ procedure enqueues the current dequeued stream object by updating its time interval and bandwidth (lines 19 to 27). Once the value of the variable *totalCurrentBandwidth* has been updated, the main procedure generates the final stream tuple values as an output, using the procedure $G_{ET}D_{ISTRIBUTION}V_{ALUE}$ (lines 15 to 17). For each call of the procedure

PROCEDURE $S_{TREAM}G_{ENERATOR}$

```
1: totalCurrentBandwidth ← 0
2: timeInChosenUnit ← 0
3: on ← false
4: d ← GETDISTRIBUTIONVALUE()
5: for i ← 1 to N do
6:    PriorityQueue.enqueue(d, bandwidth ← Math.power(2,i), timeInChosenUnit ←
      currentTime())
7: end for
8: current ← PriorityQueue.dequeue()
9: while (true) do
10:   wait(oneStep)
11:   if (currentTime() > current.timeInChosenUnit) then
12:      current ← PriorityQueue.dequeue()
13:      SWAPSTATUS(current)
14:   end if
15:   for j ← 1 to totalCurrentBandwidth do
16:      OUTPUT GETDISTRIBUTIONVALUE()
17:   end for
18: end while
```

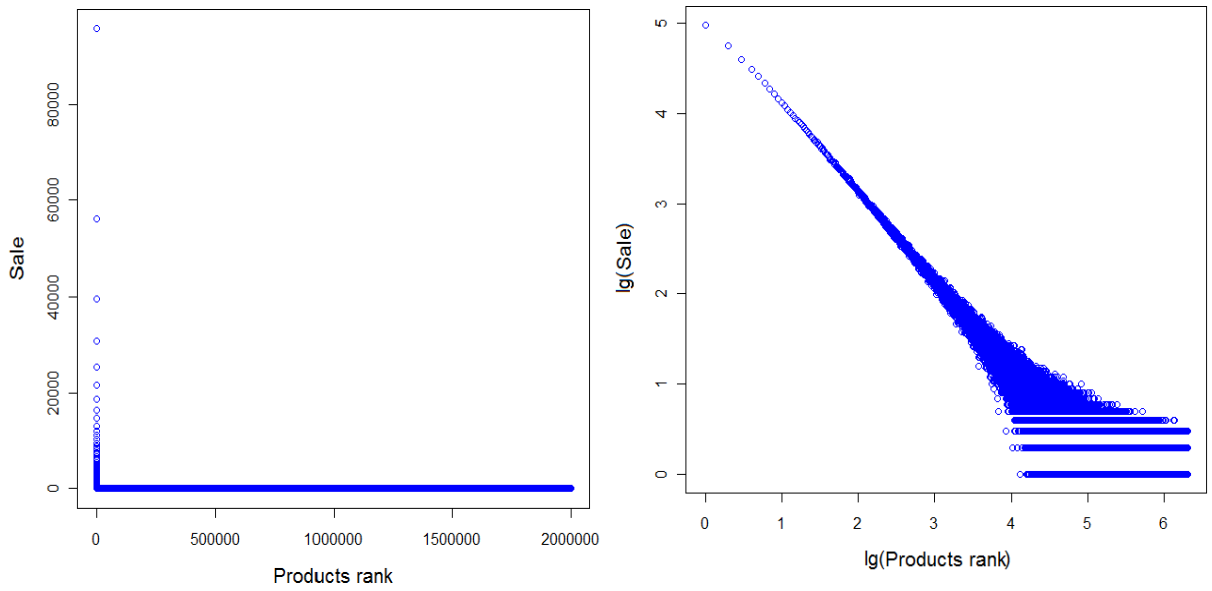
PROCEDURE $S_{WAP}S_{TATUS}(current)$

```
19: timeInChosenUnit ← (current.timeInChosenUnit + getNextRandom()
    ×oneStep × currentBandwidth)
20: if on then
21:    totalCurrentBandwidth ← totalCurrentBandwidth - current.bandwidth
22:    on ← false
23: else
24:    totalCurrentBandwidth ← totalCurrentBandwidth + current.bandwidth
25:    on ← true
26: end if
27: PriorityQueue.enqueue(current)
```

PROCEDURE $G_{ET}D_{ISTRIBUTION}V_{ALUE}()$

```
28: sumOfFrequency ← ∫ 1/x dx at x=max - ∫ 1/x dx at x=min
29: random ← getNextRandom()
30: distributionValue ← inverseIntegralOf(random × sumOfFrequency + ∫ 1/x dx at x=min)
31: RETURN ⌊distributionValue⌋
```

Figure 5.4: Pseudo-code for benchmark



(a) On plain scale (b) On log scale

Figure 5.5: A distribution using Zipf's law

$GET\,DISTRIBUTION\,VALUE$, it returns the random value by implementing Zipf's law(lines 28 to 31).

The experimental representation of the benchmark is shown in Figure 5.5 and Figure 5.6, while the environment in which the experiments have been conducted is described in Section 5.4.1. As described previously in this section, the benchmark is based on two characteristics; one is the frequency of sales of each product while the other is the flow of these sales transactions. Figure 5.5 validates the first characteristic i.e. Zipfian distribution for market sales. In the figure the *x-axis* represents the variety of products while the *y-axis* represents the sales. It can be observed that only a limited number of products (20%) are sold frequently while the rest of the products are sold rarely.

The HYBRIDJOIN algorithm is adapted to these kinds of benchmarks in which only a small portion of R is accessed again and again while the rest of R is accessed rarely.

Figure 5.6 represents the flow of transactions, which is the second characteristic of the benchmark. It is clear that the flow of transactions varies with time and the stream is bursty rather than input appearing at a regular rate.

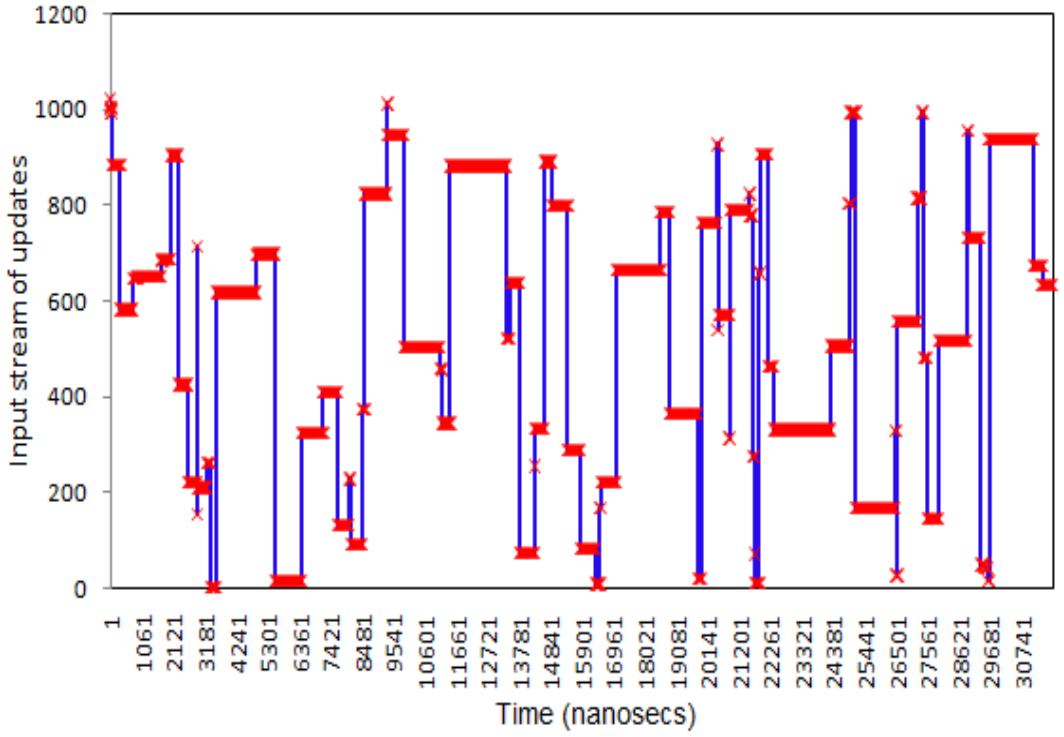


Figure 5.6: An input stream with bursty and self-similarity characteristics

5.4 Experiments

An extensive experimental evaluation of the HYBRIDJOIN has been performed on the basis of synthetic data set. This section illustrates the environment of the experiments and analyses the results that are obtained using different scenarios.

5.4.1 Experimental Setup

In order to implement the prototypes of the existing MESHJOIN, Index Nested Loop Join(INLJ) and proposed HYBRIDJOIN the same hardware and measurement strategy have been used as in R-MESHJOIN. However the data specifications are different.

The performance of each of the algorithms has been analysed using synthetic data. In the case of R-MESHJOIN (in Chapter 4) R was stored in a text file while in HYBRIDJOIN, since index is used on R, the master data R has been stored on disk using a *MySQL version 5.0* database. The bursty type of stream data has been generated at run time using the benchmark algorithm that was described previously.

In the transformation, a join is normally performed between the *primary key* (a key in R) and the *foreign key* (a key in the stream tuple) and therefore HYBRIDJOIN supports

Table 5.3: Data specification

Parameter	**value**
Memory	
Total allocated memory M	50MB to 250MB
Exponent	
Zipfian exponent value e	0 to 1
Master data	
Size of the master data R	0.5 *millions* to 8 *millions tuples*
Size of each tuple	120 *bytes*
Stream data	
Size of each tuple	20 *bytes*
Size of each node in the queue	12 *bytes*
Stream arrival rate λ	125 to 2000 *tuples/sec*
Benchmark	
Based on	Zipf's law
Characteristics	Bursty and self-similar

joins for both one-to-one and one-to-many relationships. In order to implement the join for one-to-many relationships it is necessary to store multiple values in the hash table against one key value. However the hash table provided by the *Java API* does not support this feature. Therefore, *Multi-Hash-Map*, provided by *Apache*, has been used as the hash table in the experiments. The detailed specification of the data set that has been used for the analysis is shown in Table 5.3. The performance of HYBRIDJOIN has been compared with that of MESHJOIN and INLJ while varying the total allocated memory M, the size of R on disk, the value of the Zipfian exponent, and the stream arrival rate λ. However, the other parameters, such as the size of the stream buffer, the size of each disk tuple, the size of each stream tuple, and the size of each node in the queue are considered fixed. The stream data set used to evaluate HYBRIDJOIN here is based on Zipf's law and has two important characteristics, bursty and self-similarity which are described in Section 5.3. The performance of all the algorithms has been tested by varying the Zipfian exponent value from 0 to 1.

5.4.2 Experimental Results

The experiments have been conducted in two dimensions. First dimension covers the performance evaluation of all three approaches, while second dimension validates the cost by comparing it with the calculated cost.

Performance comparison

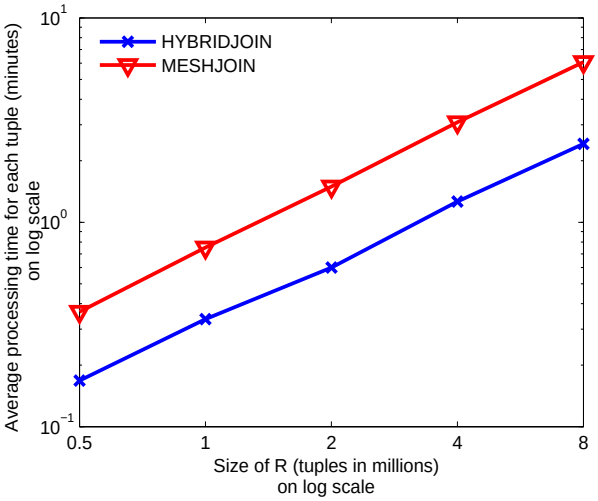
In these experiments the performance has been compared in two different ways. First, HYBRIDJOIN has been compared with MESHJOIN and INLJ with respect to time, both the *processing time* and the *waiting time*. Secondly, the performance has been compared in terms of service rate.

Performance comparisons with respect to time: To test the performance with respect to time two different types of experiments have been conducted. The experiment, shown in Figure 5.7(a), presents the comparisons with respect to the *processing time*, while Figure 5.7(b) depicts the comparisons with respect to *waiting time*. The terms *processing time* and *waiting time* have already been defined in Section 5.2. According to Figure 5.7(a) the *processing time* in the case of HYBRIDJOIN is significantly smaller than that of MESHJOIN. The reason behind this is that in HYBRIDJOIN a different strategy has been used to access R. The MESHJOIN algorithm accesses all disk partitions with the same frequency without considering the rate of use of each partition on the disk. In HYBRIDJOIN an index-based approach that never reads unused disk partitions has been implemented to access R. The experiment has not reflected the *processing time* for INLJ because it was constant even when the size of R changes.

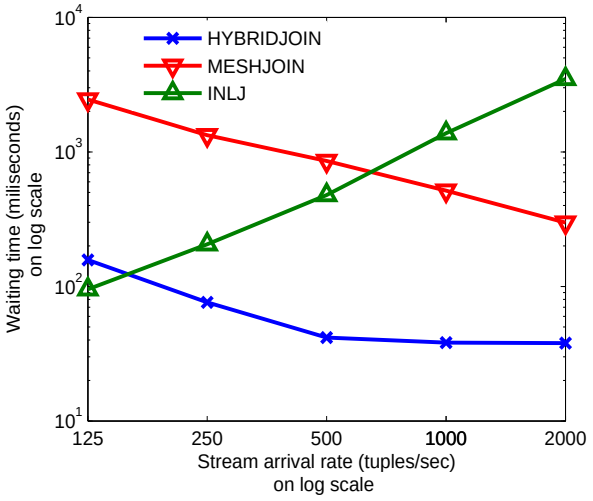
In the experiment shown in Figure 5.7(b) the time that each algorithm waits has been compared. In the case of INLJ, since the algorithm works at tuple level, the algorithm does not need to wait, but this delay then appears in the form of a stream backlog that occurs due to a faster incoming stream rate than the processing rate. The amount of this delay increases further when the stream arrival rate increases.

Turning to the other two approaches, from the figure the ratio of *waiting time* in MESHJOIN is greater than in HYBRIDJOIN. In HYBRIDJOIN, since there is no constraint to match each stream tuple with the whole of R, each disk invocation is not synchronised with the stream input. However, for stream arrival rates of less than *150 tuples/sec*, the waiting time in HYBRIDJOIN is greater than that in INLJ. A plausible reason for this is the greater *I/O* cost in the case of HYBRIDJOIN when the size of the input stream has been assumed to be equal in both algorithms.

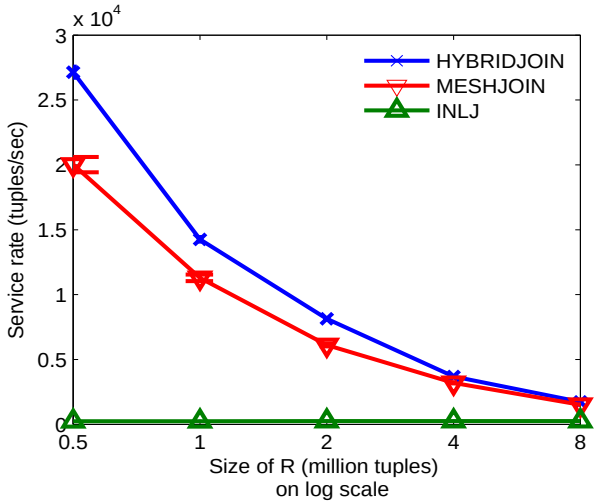
Performance comparisons with respect to service rate: In this category of experiments the performance of HYBRIDJOIN has been compared with that of the other



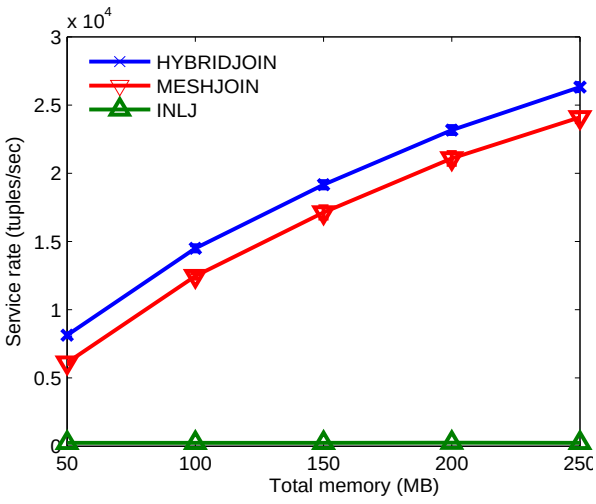
(a) Processing time (*y-axis* is on log scale)



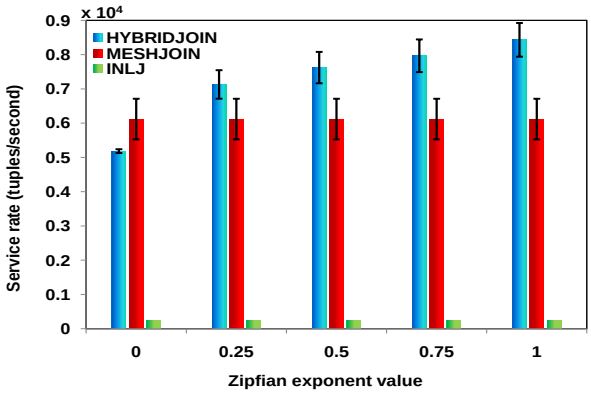
(b) Waiting time



(c) Performance comparison with 95% confidence interval while M=*50MB* and R_t varies.



(d) Performance comparison with 95% confidence interval while R_t=*2* million tuples and M varies.



(e) Performance comparison with 95% confidence interval when M=50MB, R_t=*2* million tuples while the value of Zipfian exponent varies.

Figure 5.7: Experimental results

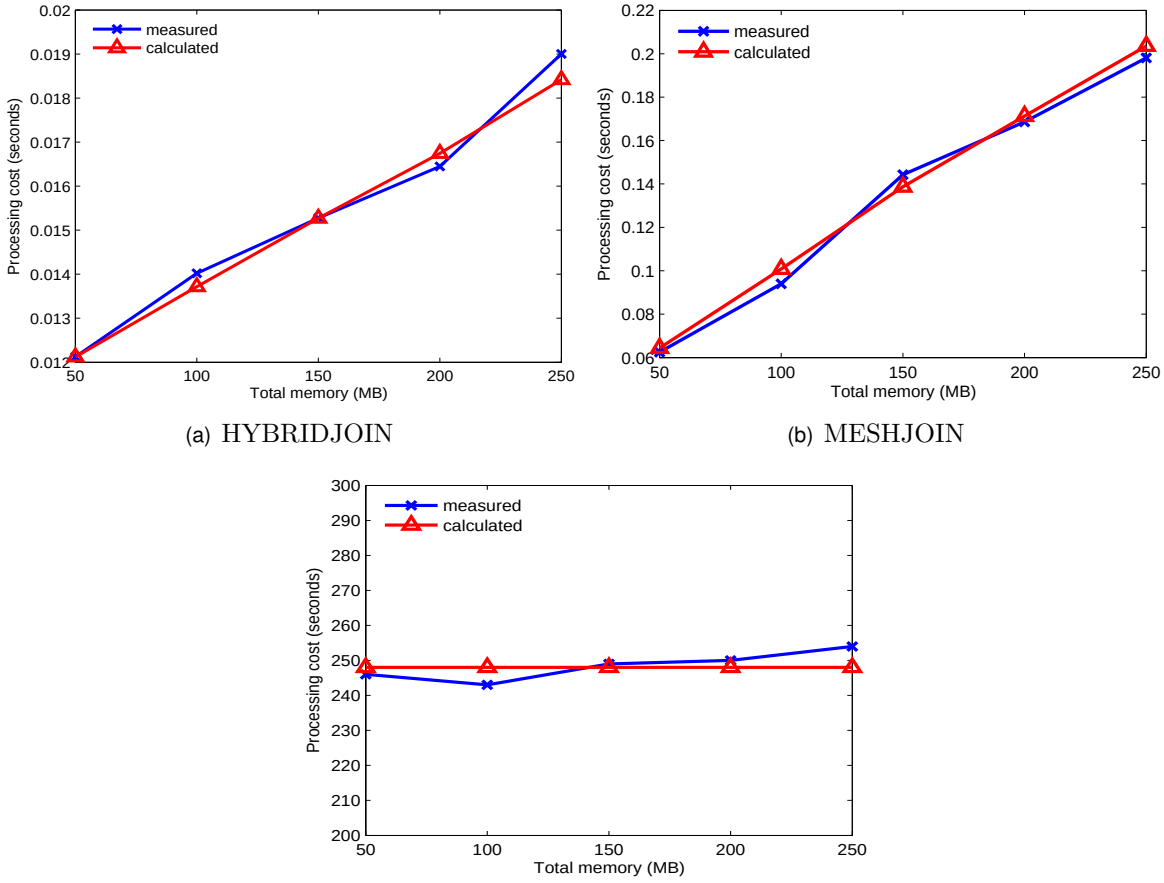
two join algorithms in terms of the service rate by varying both the total memory budget and the size of R with a bursty stream. In the experiment shown in Figure 5.7(c) the total allocated memory for the join is assumed fixed while the size of R varies exponentially. It can be observed that for all sizes of R, the performance of HYBRIDJOIN is significantly better than the other join approaches.

In the second experiment of this category the performance of HYBRIDJOIN has been analysed using different memory budgets, while the size of R is fixed (*2* million tuples). Figure 5.7(d) depicts the comparisons of all three approaches. From the figure it is clear that for all memory budgets the performance of HYBRIDJOIN is better than the other two algorithms.

Finally, the performance of HYBRIDJOIN has been evaluated by varying the skew in the input stream S. The value of the Zipfian exponent e is varied in order to vary the skew. In these experiments it was allowed to range from 0 to 1. At 0 the input stream S is uniform and the skew increases as e increases. Figure 5.7(e) presents the results of the experiment. It is clear from Figure 5.7(e) that under all values of e except 0, HYBRIDJOIN performs considerably better than MESHJOIN and INLJ. Also this improvement increases with an increase in e. The plausible reason for this better performance in the case of HYBRIDJOIN is that the algorithm does not read unused parts of R into memory and this saves unnecessary I/O cost. Moreover, when the value of e increases the input stream S becomes more skewed and, consequently, the I/O cost decreases due to an increase in the size of the unused part of R. However, as mentioned in Section 5.2.3, in a particular scenario, when e is equal to 0, HYBRIDJOIN performs worse than MESHJOIN but worse only by a constant factor.

Cost Validation

In this experiment the cost models for all three approaches are validated by comparing the calculated cost with the measured cost. Figure 5.8 presents the comparisons of both costs. In the figure it is demonstrated that the calculated cost closely resembles the measured cost in every approach, which validates the accuracy of the cost models.



(a) HYBRIDJOIN

(b) MESHJOIN

(c) INLJ

Figure 5.8: Cost validation

5.5 Summary

In this chapter two issues in the existing MESHJOIN algorithm have been addressed. One is an inefficient approach for accessing the master data and the other is dealing with intermittency in stream data. As a solution a new join algorithm, HYBRIDJOIN, has been proposed. The main objectives in HYBRIDJOIN are: (a) to minimize the stay of every stream tuple in the join window by improving the efficiency of the access to the master data, and (b) to deal with the true nature of update streams. A cost model has been developed for HYBRIDJOIN. To validate that cost model and to achieve the maximum performance within the limited resources a tuning module has also been presented for HYBRIDJOIN. A benchmark based on Zipfian distribution has been designed to test this approach. To validate the arguments a prototype of HYBRIDJOIN has been implemented that demonstrates a significant improvement in service rate under limited memory.

6

X-HYBRIDJOIN

6.1 Introduction

The HYBRIDJOIN algorithm as discussed in Chapter 5 was designed to address two particular concerns. The first objective was to amortise the disk I/O cost over the fast input data stream more effectively by introducing an index-based approach for accessing the master data R. The second objective was to deal with the bursty nature of the data stream effectively.

The HYBRIDJOIN algorithm developed here efficiently amortises the fast input stream using an index-based approach to access the master data and can deal with bursty streams. However, the performance can be improved more by considering non-uniform characteristic of market data. Current market data analysis shows that a few items in a product range are bought with higher frequency [3]. According to one market survey [3] the sales frequency of products can be formulated using the 80/20 Rule, i.e. 80% of total sales are made from only 20% of the products. To elaborate on this further, we consider Figure 5.5

shown in Chapter 5. In the figure it can be observed that the frequency of selling a small number of products is significantly higher compared to the rest of the products. Therefore, in the stream that propagates toward the warehouse, most of the tuples need to join with a small number of records on the disk again and again. Currently the HYBRIDJOIN algorithm does not consider this feature and loads pages from the disk frequently. Consider the reduction in I/O costs if these pages could be held permanently in memory. Therefore, the question remains how much potential for improvement remains untapped in HYBRIDJOIN because the algorithm does not cache the master data.

These considerations have motivated the proposal of an extension of HYBRIDJOIN, called X-HYBRIDJOIN (Extended Hybrid Join) [84][1]. The key difference between the two is that in X-HYBRIDJOIN the algorithm stores in memory the portion of the master data which matches the frequent items in the stream. This reduces the I/O cost substantially, which improves the performance of the algorithm.

The ideal approach for obtaining high performance for the algorithm would be to fit all of that 20% of R into the memory permanently. But this is impractical because of the large size of R and the fact that there would then be less memory available to execute the join operator. In these kinds of situations a smaller portion of R is normally loaded into memory, as shown in Figure 6.1. In the figure the total master data R is divided into two parts. One is 20% of R while the other is 80% of R. If the 20% of R (i.e. $0.2R$) is still bigger than the non-swappable part then again $0.2R$ is divided into two parts containing 20% and 80%. This practice will continue until 0.2^nR becomes less or equal to the size of the non-swappable part. Initially the size of each swappable and non-swappable part are set to be equal to one disk partition of size d, which is equal to the size of the disk buffer in HYBRIDJOIN. However, in the latter part of this chapter we perform tuning in order to determine the optimal memory size for each component of the algorithm.

The rest of the chapter is structured as follows. Section 6.2 presents the memory architecture, algorithm, and cost model for the proposed X-HYBRIDJOIN. The experimental study is discussed in Section 6.3. In Section 6.4 the tuning module for X-HYBRIDJOIN is presented. Section 6.5 compares the performance results before and after the tuning of X-HYBRIDJOIN. Finally Section 6.6 presents a summary of the chapter.

[1]This work has been published in 28th British National Conference on Databases (BNCOD'11).

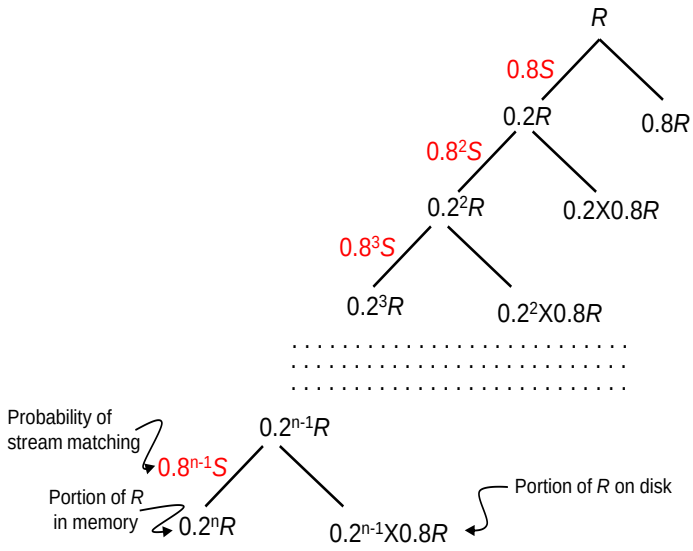


Figure 6.1: A general sketch of the classification of R into non-swappable and swappable parts

6.2 X-HYBRIDJOIN

This section describes the memory architecture, pseudo-code and cost model for an enhanced version of HYBRIDJOIN, called X-HYBRIDJOIN.

6.2.1 Memory Architecture

The execution architecture for X-HYBRIDJOIN is shown in Figure 6.2. In the X-HYBRIDJOIN algorithm the disk buffer is divided into two parts. One part stores the most popular part of the master data R in memory permanently; this is called the non-swappable part of the disk buffer. The other part of the disk buffer is swappable and is used to load the rest of R into memory in the same way as in HYBRIDJOIN. Initially, the same amount of memory is assigned to both parts. The role of other components like queue Q, hash table H and the stream buffer is quite similar to that in HYBRIDJOIN. Similarly to HYBRIDJOIN, it is also assumed here that R is sorted with respect to the access frequency and has an index with the join attribute as the key. The main objective of the X-HYBRIDJOIN algorithm is that, for each iteration when the disk partition is loaded into the swappable part of the disk buffer, apart from matching with that partition as in HYBRIDJOIN, the algorithm matches both parts of the disk buffer with all the stream tuples available in memory without any extra disk I/O cost. This additional feature amortises the fast arrival stream by minimizing the disk I/O cost.

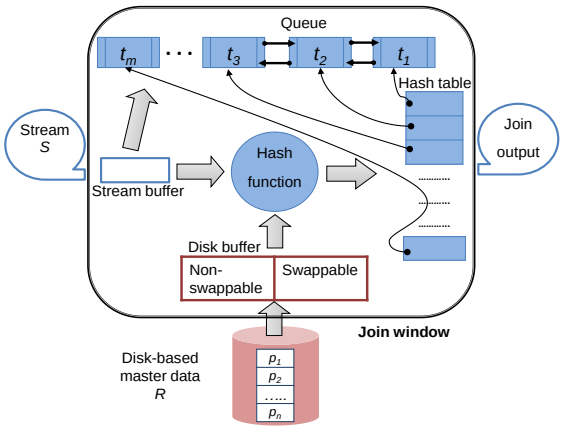


Figure 6.2: Architecture of X-HYBRIDJOIN

6.2.2 Algorithm

Once the available memory has been distributed among the join components, the algorithm is ready to execute according to the procedure described in Algorithm 4. Before starting the actual join execution, the algorithm reads a particular portion of the master data R into the non-swappable part of the disk buffer (line 1). Similarly to HYBRIDJOIN, in the beginning all slots in the hash table H are empty; therefore, h_S is assigned to w (line 2). In the abstract level description, the algorithm contains two kinds of loops. One is called the outer loop, which is an endless loop (line 3). The key objective of the outer loop is to build the stream in the hash table. Under the outer loop, the algorithm runs two independent inner loops. One loop implements the probing module for the non-swappable part of the disk buffer, while the other inner loop implements the probing of the swappable part of the disk buffer. As the outer loop begins, the algorithm observes the status of the stream buffer. If stream input is available the algorithm reads the w tuples from the stream buffer and loads them into the hash table, while also enqueuing their attribute values into the queue. After completing the stream input the algorithm resets w to 0 (line 4-7). The algorithm then executes the first inner loop, in which it reads all tuples one-by-one from the non-swappable part of the disk buffer and looks them up in the hash table. In the case of a match, the algorithm generates the join output. Due to the multi-hash-map, there can be more than one match against one disk tuple. After generating the join output the algorithm deletes all matched tuples from the hash table, along with the corresponding nodes from the queue. The algorithm also increments w with the number of vacated slots in the hash table (line 8-14). Before starting the second

Algorithm 4 X-HYBRIDJOIN

Input: A master data R with an index on join attribute and a stream of updates S
Output: $S \bowtie R$
Parameters: w tuples of S and a partition p_i of R
Method:

```
1: LOAD first partition p_1 of R into the non-swappable part of the disk buffer.
2: w ← h_S
3: while (true) do
4:   if (stream available) then
5:     READ w tuples from the stream buffer, load them into H and enqueue their join
       attribute values into Q.
6:     w ← 0
7:   end if
8:   for each tuple r in p_1 do
9:     if r ∈ H then
10:      OUTPUT r ⋈ H
11:      w ← w+number of matching tuples found in H
12:      DELETE all matched tuples from H and the corresponding nodes from Q.
13:    end if
14:  end for
15:  READ the oldest join attribute value from Q.
16:  LOAD a disk partition p_i (where 2 ≤ i ≤ n)of R into the swappable part of the
       disk buffer using the join attribute value as an index.
17:  for each tuple r in p_i do
18:    if r ∈ H then
19:      OUTPUT r ⋈ H
20:      w ← w+number of matching tuples found in H
21:      DELETE all matched tuples from H and the corresponding nodes from Q.
22:    end if
23:  end for
24: end while
```

inner loop, the algorithm reads the oldest value of the join attribute from the queue and loads a disk partition p_i (where $2 \leq i \leq n$) into the swappable part of the disk buffer, using that join attribute value as an index (line 15, 16). As the specified disk partition is loaded into the swappable part of the disk buffer, the algorithm starts the second inner loop and repeats all the steps described in the first inner loop (line 17-23).

6.2.3 Cost Model

X-HYBRIDJOIN uses one more component than HYBRIDJOIN. Therefore a new cost model for X-HYBRIDJOIN is presented here. By adopting a similar cost calculation

approach, as described for the earlier algorithms, separate formulas for the calculation of memory and processing time are derived here. Equation 6.1 describes the total memory used to implement the algorithm except for the stream buffer, whereas Equation 6.2 calculates the processing cost for w tuples. The symbols used to measure the cost have already been specified partially in Chapter 3 and Chapter 4.

Memory Cost

In X-HYBRIDJOIN, the disk buffer is divided into two equal parts. One is swappable, the other is non-swappable. As noted above, the largest share of the total memory is used for the hash table; a much smaller portion is used for the disk buffer. The queue size is a constant fraction of the hash table size. The memory for each component of X-HYBRIDJOIN can be calculated as shown below.

Memory reserved for the swappable and non-swappable part of the disk buffer = $v_P + v_P = 2v_P$

Memory reserved for the hash table= $\alpha(M - 2v_P)$

Memory reserved for the queue= $(1 - \alpha)(M - 2v_P)$

The total memory used by X-HYBRIDJOIN can be determined by aggregating all of the above.

$$M = 2v_P + \alpha(M - 2v_P) + (1 - \alpha)(M - 2v_P) \tag{6.1}$$

Currently the memory reserved by the stream buffer is not included because of its small size.

Processing Cost

This section calculates the processing cost for X-HYBRIDJOIN. The cost for one loop iteration of the algorithm is denoted by c_{loop} and expressed as the sum of the costs for the individual operations. To make it simple, the processing cost for each component is calculated separately first.

Cost to read swappable or non-swappable parts of the disk buffer= $c_{I/O}(v_P)$

Cost to look-up swappable and non-swappable parts of the disk buffer in the hash table= $2d \cdot c_H$ where d is the size of each part in tuples.

Cost to generate the output for w matching tuples= $w \cdot c_O$

Cost to remove w tuples from the hash table and the queue= $w \cdot c_E$

Cost to read w tuples from stream S= $w \cdot c_S$

Cost to append w tuples into the hash table and the queue= $w \cdot c_A$

As the non-swappable part of the disk buffer is read only once before execution starts, this is excluded. By aggregating the terms, the total cost for one loop iteration is:

$$c_{loop}(secs) = 10^{-9}[c_{I/O}(v_P) + 2d.c_H + w(c_O + c_E + c_S + c_A)] \tag{6.2}$$

In c_{loop} seconds the algorithm processes w tuples of stream S; therefore, the service rate μ can be calculated by dividing w by the cost for one loop iteration, as shown in Equation 6.3.

$$\mu = \frac{w}{c_{loop}} \tag{6.3}$$

6.3 Experimental Results

The experimental study analysed the results from two different perspectives. The results presented in Section 6.3.1 compare the performance of both algorithms while the results in Section 6.3.2 focus on the role of the non-swappable part of the disk buffer in stream processing. The setup used for these experiments has already been described in Chapter 5.

6.3.1 Performance Comparisons

The two possible parameters that can vary and directly affect the performance of an algorithm are the total memory available for the algorithm and the size of the master data. In the current experiments the algorithm has been tested for different values of these parameters and the performance has been compared with HYBRIDJOIN at each specification. As an example, to clarify this comparison further the performance of HYBRIDJOIN has also been tested by keeping the disk buffer size equal to the size of the swappable plus non-swappable part of the disk buffer in X-HYBRIDJOIN.

Performance comparisons when the size of the master data varies: In the experiment shown in Figure 6.3(a), it has been assumed that total allocated memory for the join is fixed while the size of the master data R is increased exponentially. Figure 6.3(a) shows that for all sizes of R, the performance of X-HYBRIDJOIN is substantially better

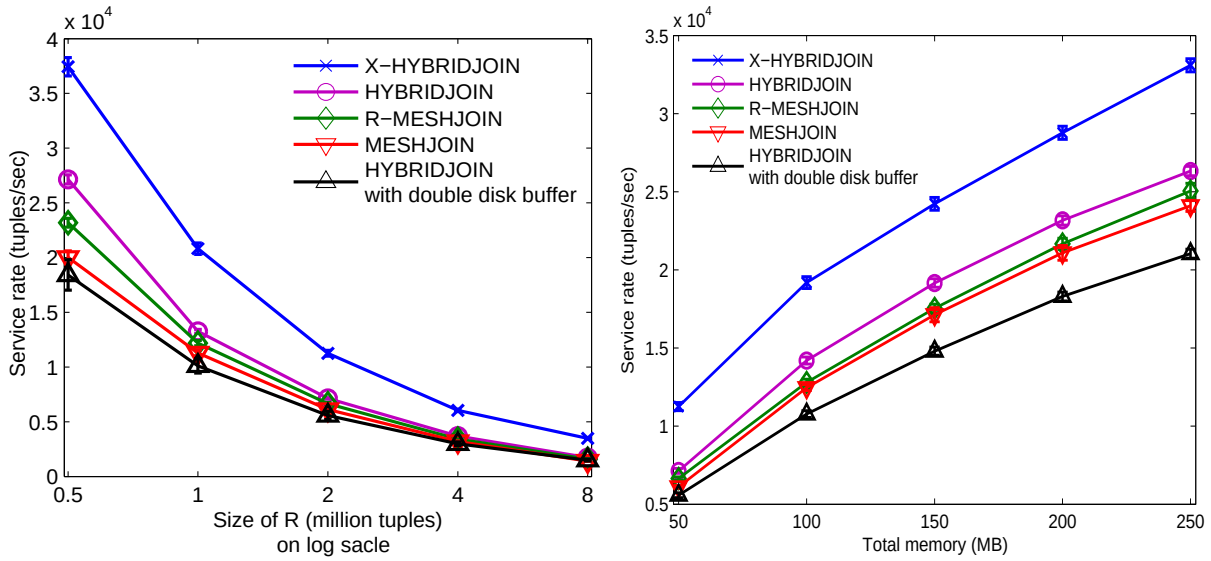
than all the other approaches. Another key observation is that when R is 0.5 million the performance of HYBRIDJOIN is almost 70% of X-HYBRIDJOIN, but when R is equal to 8 million this percentage decreases to 50%. This means that, compared to X-HYBRIDJOIN, the performance of HYBRIDJOIN decreases more sharply as R increases.

Performance comparisons when the size of available memory varies: In our second experiment, the performance of X-HYBRIDJOIN has been analysed using different memory budgets while the size of R is fixed (2 million tuples). Figure 6.3(b) presents the results of the experiment. The figure indicates that, for all memory budgets, the performance of X-HYBRIDJOIN is better than all the other algorithms. The reason behind this improvement is the addition of a non-swappable part in the component disk buffer. In our calculations, introducing the non-swappable part into X-HYBRIDJOIN can save about 33% of the disk I/O cost. Although keeping the non-swappable part in memory increases the look-up cost and reduces the memory available for the hash table, both these factors are very small compared to the other significant factor, the disk I/O cost.

As an example and for further satisfaction the performance of HYBRIDJOIN has also been tested while keeping the size of the disk buffer equal to the size of the X-HYBRIDJOIN disk buffer. The performance results in both cases, i.e. when the size of R and the total memory budget change, are shown in Figure 6.3(a) and Figure 6.3(b). In both figures, it can be observed that the performance under these settings is even worse than HYBRIDJOIN with optimal disk buffer size. The plausible reason for this behaviour may be that, once the optimal size of the disk buffer has been reached, a further increase does not increase the stream-matching probability anymore after getting the maximum, while on the other hand it increases the disk I/O cost and the look-up cost. In addition it also reduces the memory budget for the hash table.

6.3.2 Role of the Non-swappable Part in Stream Processing

As described in Section 6.1, a large number of tuples in the stream belongs to a small number of products; therefore, to join with the master data, the relative part of the master data is required to be brought into the memory again and again. To observe this factor more closely an experiment has been performed that accumulates the stream tuples



(a) Performance comparison with 95% confidence interval while M= *50MB* and R_t varies.

(b) Performance comparison with 95% confidence interval while R_t=*2* million tuples and M varies.

Figure 6.3: Experimental results: HYBRIDJOIN vs X-HYBRIDJOIN

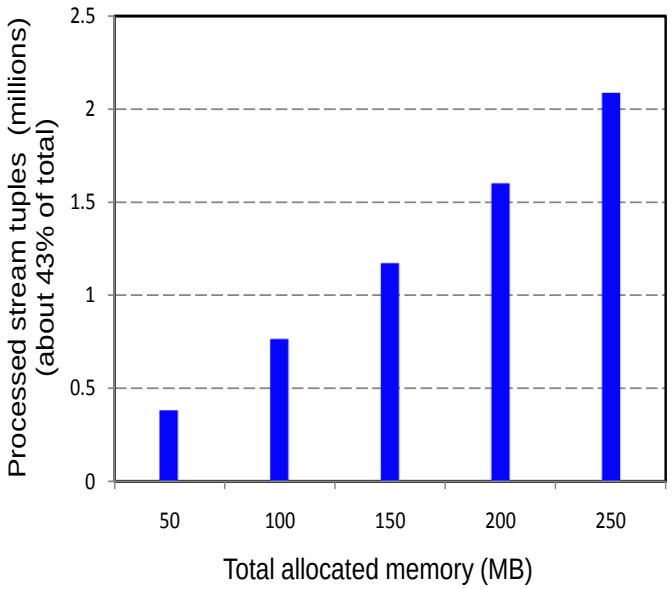


Figure 6.4: Total number of stream tuples processed with the non-swappable part of the disk buffer in 4000 iterations

which are processed using only the non-swappable part of the disk buffer. The results of this experiment are shown in Figure 6.4. As in the experimental settings, the size of the non-swappable part has been set to be equal to the size of the swappable part. It is clear from the figure, that in 4000 iterations when the memory budget is 50 MB and the size of R is 2 million tuples, about 0.4 million stream tuples are processed through the non-swappable part of the disk buffer. This ratio increases when the total allocated

memory is increased. For 250 MB memory with the same size of R (which is 2 million) tuples, the number of tuples processed reaches more than 2 million. In HYBRIDJOIN, since this non-swappable part is loaded from the disk each time, the I/O cost is increased significantly.

6.4 Tuning

The stream-based join operators normally execute within limited memory resources where a number of operations are executing in parallel. Therefore, to avoid the blocking of these operations, a join operator should not be resource-intensive. The tuning process, which is also an application of the cost model, is used to optimise the performance of the algorithm under resource constraints. Normally if these kinds of algorithms are seen in isolation, having more memory available would be better for each component. However, assuming a fixed memory allocation provides a trade-off in the distribution of memory. Assigning more memory to one component means that less memory is available for other components. Therefore it is necessary to find the optimal distribution of memory among all components in order to attain maximum performance. A very important component here is the disk buffer because reading data from disk to memory is very expensive.

X-HYBRIDJOIN minimizes the disk access cost and improves performance significantly by introducing the non-swappable part of the disk buffer. But in X-HYBRIDJOIN the memory assigned to the swappable part of the disk buffer is equal to the size of the disk buffer in HYBRIDJOIN and the same amount of memory is allocated to the non-swappable part of the disk buffer. In the following it will be shown that this is not optimal. Therefore, in this section the tuning of the X-HYBRIDJOIN is performed to assign the optimal amount of memory among the components of the algorithm [87][2].

6.4.1 Revised Cost Model

In order to tune X-HYBRIDJOIN, it is first necessary to revise the cost model. The reason for this revision is that X-HYBRIDJOIN uses equal memory for both the swappable and the non-swappable parts of the disk buffer, and therefore the formulas do not apply for

[2]This work has been published in International Multi-topic Conference (IMTIC'12).

other relative sizes. Using a revised cost model, Equation 6.4 describes the total memory used to implement the algorithm, while Equation 6.5 calculates the processing cost for w tuples.

Memory cost: Since the optimal values for the sizes of both the swappable part and non-swappable part can be different, k number of pages is assumed for the swappable part and l number of pages for the non-swappable part. The memory for each component can be calculated as given below:
Memory for the swappable part of the disk buffer (bytes)= $k \cdot v_P$
Memory for the non-swappable part of the disk buffer (bytes)= $l \cdot v_P$
Memory for the hash table (bytes)= $\alpha[M - (k + l)v_P]$
Memory for the queue (bytes)= $(1 - \alpha)[M - (k + l)v_P]$
The total memory used by the algorithm can be determined by aggregating the above.

$$M = (k + l)v_P + \alpha[M - (k + l)v_P] + (1 - \alpha)[M - (k + l)v_P] \tag{6.4}$$

The memory reserved by the stream buffer is not included due to its small size.

Processing cost: This section revises the processing cost for X-HYBRIDJOIN. The cost for one iteration of the algorithm is denoted by c_{loop} and expressed as the sum of the costs for the individual operations. The processing cost for each component is calculated separately first.
Cost to read the non-swappable part of the disk buffer (nanoseconds) = $c_{I/O}(l \cdot v_P)$
Cost to read the swappable part of the disk buffer (nanoseconds)= $c_{I/O}(k \cdot v_P)$
Cost to look-up the non-swappable part of the disk buffer in the hash table (nanoseconds) = $d_N c_H$ where $d_N = l\frac{v_P}{v_R}$ is the size of the non-swappable part of the disk buffer in terms of tuples.
Cost to look-up the swappable part of the disk buffer in the hash table (nanoseconds)= $d_S c_H$ where $d_S = k\frac{v_P}{v_R}$ is the size of the swappable part of the disk buffer in terms of tuples.
Cost to generate the output for w matching tuples (nanoseconds)= $w \cdot c_O$
Cost to delete w tuples from the hash table and the queue (nanoseconds)= $w \cdot c_E$
Cost to read w tuples from stream S into the stream buffer (nanoseconds)= $w \cdot c_S$
Cost to append w tuples into the hash table and the queue (nanoseconds)= $w \cdot c_A$

As the non-swappable part of the disk buffer is read only once before the actual execution starts, it is excluded. The total cost for one loop iteration is:

$$c_{loop}(\text{secs}) = 10^{-9}[c_{I/O}(k \cdot v_P) + (d_N + d_S)c_H + w(c_O + c_E + c_S + c_A)] \tag{6.5}$$

If the algorithm processes w tuples in c_{loop} seconds then the service rate μ can be calculated using Equation 6.6.

$$\mu = \frac{w}{c_{loop}} \tag{6.6}$$

Once the cost has been calculated, the algorithm can be tuned on the basis of this cost model. In the following two subsections the algorithm is tuned using both empirical and mathematical approaches. Finally, the tuning results obtained in both approaches are compared to validate our cost model.

6.4.2 Tuning using Empirical Approach

This section focuses on obtaining samples for the approximate tuning of the key components. The performance is a function of two variables, the size of the swappable part of the disk buffer, d_S, and the size of the non-swappable part of the disk buffer, d_N. The performance of the algorithm has been tested for a grid of values for both components, i.e. for each setting of d_S the performance was measured against a series of values for d_N. The performance measurements for the grid of d_S and d_N are shown in Figure 6.5. The figure shows that the performance increases rapidly as the size for the non-swappable part increases. After reaching a particular value for the size of the non-swappable part, the performance starts decreasing. The plausible reason behind this behavior is that in the beginning when the size for the non-swappable part increases, the probability of matching stream tuples with disk tuples also increases and that improves the performance. But when the size for the non-swappable part is increased further it does not make a significant difference in stream-matching probability due to the factor of skew in distribution. The higher look-up cost associated with the increased non-swappable part and the fact that less memory is available for the hash table means that the performance gradually decreases.

A similar behavior has been observed when the performance has been tested for the

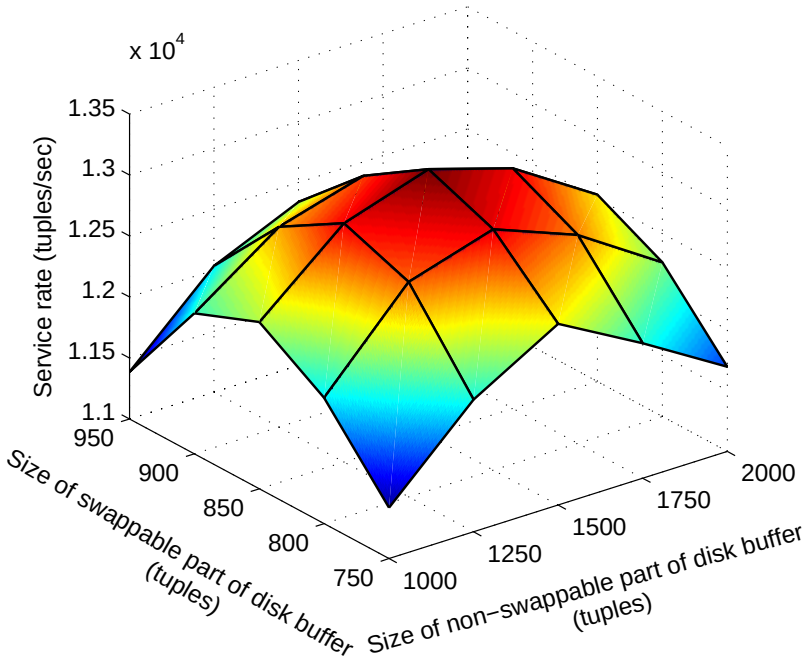


Figure 6.5: Tuning of X-HYBRIDJOIN using measurement approach

swappable part. Initially the performance increases, since the costly disk access is amortised for a larger number of stream tuples. This effect is of crucial importance, because it is this gain that gives the algorithm an advantage over a simple index-based join. It is here that the hash table is used in order to match more tuples than just the one that was used to determine the partition that was loaded. After attaining a maximum, the performance decreases because of the increase in I/O cost for loading more of R at one time in a non-selective way.

From the measurements shown in Figure 6.5 it is possible to approximate the optimal settings for both the swappable and the non-swappable parts by considering the intersection of the values of both components at which the algorithm individually performs at a maximum.

6.4.3 Tuning using Mathematical Approach

A mathematical model for the tuning is also derived based on the cost model presented in Section 6.4.1. From Equation 6.6 it is clear that the service rate depends on the size of w and the cost c_{loop}. To determine the optimal settings it is first necessary to calculate the size of w. The value of w in X-HYBRIDJOIN is the total number of stream tuples which match with both the swappable and non-swappable parts in each iteration.

Before deriving a mathematical formula for w the main components that can affect w are discussed. The main components on which the value of w depends are listed below.

a. Size of non-swappable part, d_N

b. Size of swappable part, d_S

c. Size of the master data, R_t

d. Size of hash table, h_S

Typically the stream of updates can be approximated through Zipfs law with a certain exponent value. Therefore, a significant part of the stream is joined with the non-swappable part of the disk buffer. Hence, if the size of the non-swappable part (i.e. d_N) is increased, more stream tuples will match as a result. But the probability of matching does not increase at the same rate as increasing d_N because, according to Zipfian distribution, the matching probability for the second tuple in R is half of that for the first tuple and similarly the matching probability for the third tuple is one third of that for the first tuple and so on [3, 66]. Due to this property, the size of R (denoted by R_t) also affects the matching probability. The swappable part of the disk buffer deals with the rest of the master data denoted by R' (where $R' = R_t - d_N$), which is less frequent in the stream than that part which exists permanently in memory. The algorithm reads R' in partitions, where the size of each partition is equal to the size of the swappable part of the disk buffer d_S. In each iteration the algorithm reads one partition of R' using an index on join attribute and loads it into memory through a swappable part of the disk buffer. In the next iteration the current partition in memory is replaced by a new partition, and so on. As mentioned earlier, using the Zipfian distribution the matching probability for every next tuple is less than the previous one. Therefore, the total number of matches against each partition is not the same. This is explained further in Figure 6.6, where n total partitions are considered in R'. From the figure it can be seen the matching probability for each disk partition decreases continuously as we move toward the end position in R. The size of the hash table is another component that affects w. The reason is simple: if there are more stream tuples in memory, the number of matches will be greater and vice versa. Before driving the formula to calculate w it is first necessary to understand the working

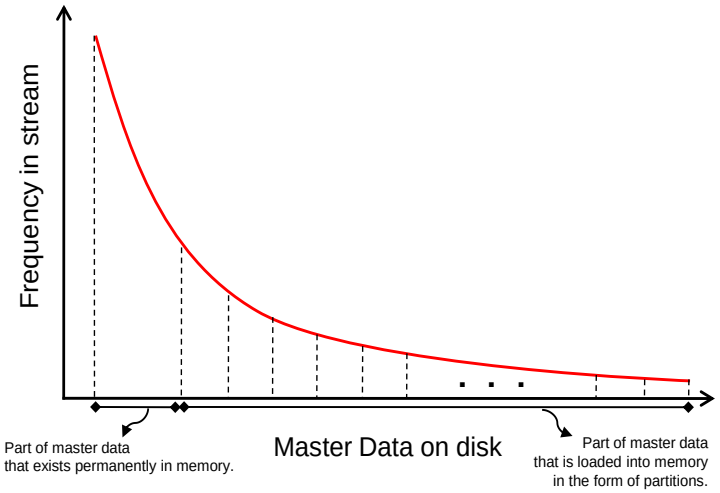


Figure 6.6: A sketch of matching probability of R in stream

strategy of X-HYBRIDJOIN. Consider for a moment that the queue contains stream tuples instead of just join attribute values. It has already been stated in Section 6.2.2 that X-HYBRIDJOIN uses two independent inner loops under one outer loop. After the end of the first inner loop, which means after finishing the processing of the non-swappable part, the queue only contains those stream tuples which are related to only the swappable part of R, denoted by R'. For the next outer iteration of the algorithm these stream tuples in the queue are considered to be an old part of the queue. In that next outer iteration the algorithm loads some new stream tuples into the queue and these new stream tuples are considered to be a new part of the queue. The reason for dividing the queue into two parts is that the matching probability for both parts of the queue is different. The matching probability for the old part of the queue is denoted by p_{old} and it is only based on the size of the swappable part of R i.e. R'. On the other hand, the matching probability for the new part of the queue, known as p_{new}, depends on both the non-swappable as well as the swappable parts of R. Therefore, to calculate w we first need to calculate both these probabilities.

Therefore, if the stream of updates S obeys Zipf's law, then the matching probability for any swappable partition k with the old part of the queue can be determined

mathematically as shown below.

$$p_k = \frac{\sum_{x=d_N+(k-1)d_S+1}^{d_N+kd_S} \frac{1}{x}}{\sum_{x=d_N+1}^{R_t} \frac{1}{x}}$$

Each summation in the above equation generates a harmonic series, which can be summed up using the formula [2] $\sum_{x=1}^{k} \frac{1}{x} = \ln k + \gamma + \varepsilon_k$, where γ is a Euler's constant [2] whose value is approximately equal to 0.5772156649 and ε_k is another constant which is $\approx \frac{1}{2k}$. The value of ε_k approaches 0 as k goes to ∞ [2]. In our case the value of $\frac{1}{2k}$ is small and therefore, it is ignored.

If there are n partitions in R', then the average probability of an arbitrary partition of R' matching the old part of the queue can be determined using Equation 6.7.

$$\overline{p}_{old} = \frac{\sum_{k=1}^{n} p_k}{n} = \frac{1}{n} \tag{6.7}$$

Now the probability of matching is determined for the new part of the queue. Since the new input stream tuple can match either the non-swappable or the swappable part of R, the average matching probability of the new part of the queue with both parts of the disk buffer can be calculated using Equation 6.8.

$$\overline{p}_{new} = p_N + \frac{1}{n} p_S \tag{6.8}$$

where p_N and p_S are the probabilities of matching for a stream tuple with the non-swappable part and the swappable part of the disk buffer respectively. The values of p_N and p_S can be calculated as below.

$$p_N = \frac{\sum_{x=1}^{d_N} \frac{1}{x}}{\sum_{x=1}^{R_t} \frac{1}{x}}$$

$$p_S = \frac{\sum_{x=d_N+1}^{R_t} \frac{1}{x}}{\sum_{x=1}^{R_t} \frac{1}{x}}$$

Assume that w are the new stream tuples that the algorithm will load into the queue in the next outer iteration. Therefore,

The size of the new part of the queue (tuples)=w

The size of the old part of the queue (tuples)=$(h_S - w)$

If w are the average number of matches per outer iteration with both the swappable and non-swappable parts in the disk buffer, then w can be calculated by applying the binomial probability distribution on Equations 6.7 and 6.8 as given below.

$$w = (h_S - w)\overline{p}_{old}(1 - \overline{p}_{old}) + w\overline{p}_{new}(1 - \overline{p}_{new})$$

After simplification the final formula to calculate w is described in Equation 6.9.

$$w = \frac{h_S\overline{p}_{old}(1 - \overline{p}_{old})}{1 + \overline{p}_{old}(1 - \overline{p}_{old}) - \overline{p}_{new}(1 - \overline{p}_{new})} \tag{6.9}$$

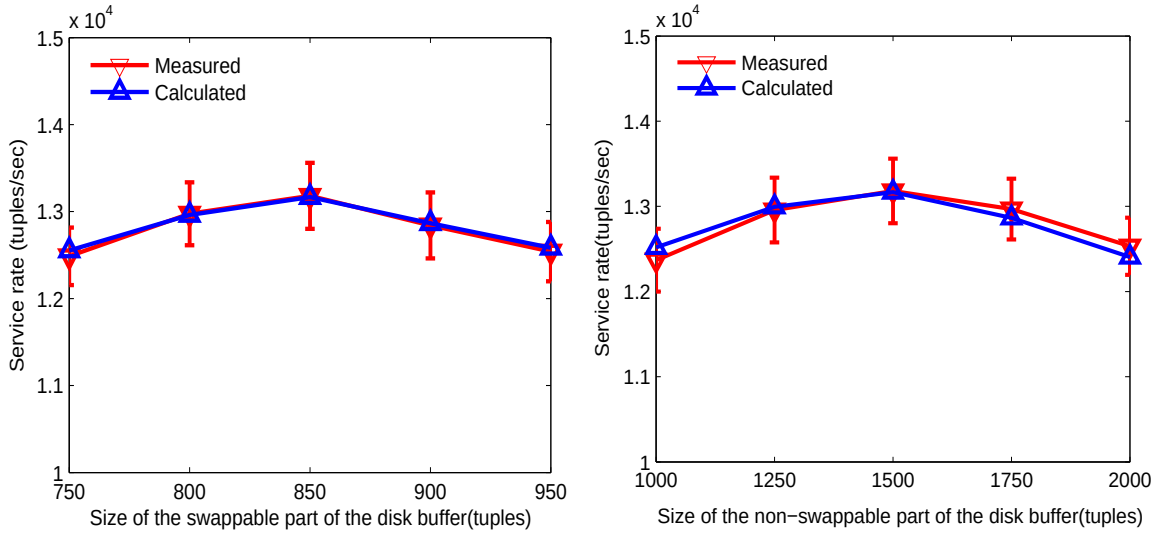
A number of experiments have been conducted to observe the effect of all necessary components on w. The results and detailed descriptions of these experiments can be found in Appendix B.

6.4.4 Comparisons of both Approaches

To validate the cost model the algorithm has been tuned using both the empirical and mathematical approaches and the results have been compared.

Swappable part: In this experiment the tuning results have been compared for the swappable part of the disk buffer using both the measurement and cost model approaches. The tuning results for each approach (with a 95% confidence interval in the case of the empirical approach) are shown in Figure 6.7(a). It is evident that at every position the results in both cases are similar, with only 0.5% deviation.

Non-swappable part: Similarly, the tuning results of both approaches have been compared for the non-swappable part of the disk buffer. The results are shown in Figure 6.7(b). Again, it is clear from the figure that the results in both cases are nearly equal, with a



(a) Tuning comparison for swappable part: based on measurements vs based on cost model

(b) Tuning comparison for non-swappable part: based on measurements vs based on cost model

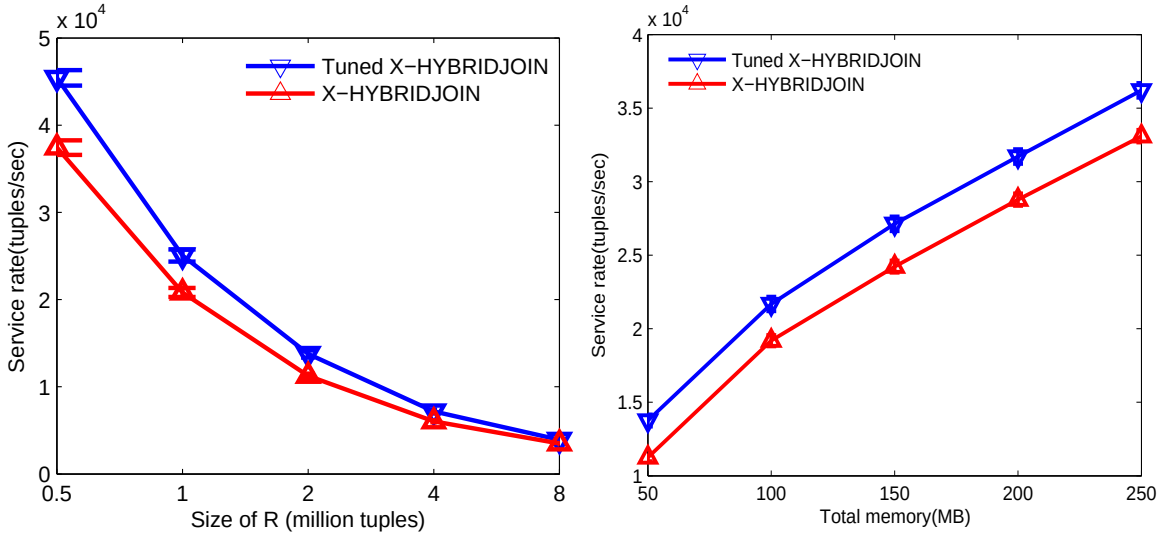
Figure 6.7: Comparisons of tuning results

deviation of only 0.6%.

6.5 Performance Evaluation after Tuning

Performance comparisons for different sizes of R: In these experiments the performance of X-HYBRIDJOIN has been compared with and without tuning by varying the size of R. It is assumed that the size of R varies exponentially while the total memory budget remains fixed (50MB) for all values of R. For each value of R the X-HYBRIDJOIN algorithm has been run both with and without optimal settings, and the performance has been measured in both cases. The performance results that have been obtained using both experiments are shown in Figure 6.8(a). It is clear that for all settings of R X-HYBRIDJOIN performed significantly better with tuning than without.

Performance comparisons for different memory budgets: In these experiments the performance in both cases has been compared using different memory budgets while the size of R is fixed (2 million tuples). Figure 6.8(b) depicts the comparisons of both cases. It can be observed that for all memory budgets the tuned X-HYBRIDJOIN again performed significantly better than simple X-HYBRIDJOIN.



(a) Performance comparison with 95% confidence interval while M= *50MB* and R_t varies.

(b) Performance comparison with 95% confidence interval while R_t=*2* million tuples and M varies.

Figure 6.8: Performance comparisons: Tuned X-HYBRIDJOIN vs X-HYBRIDJOIN without tuning

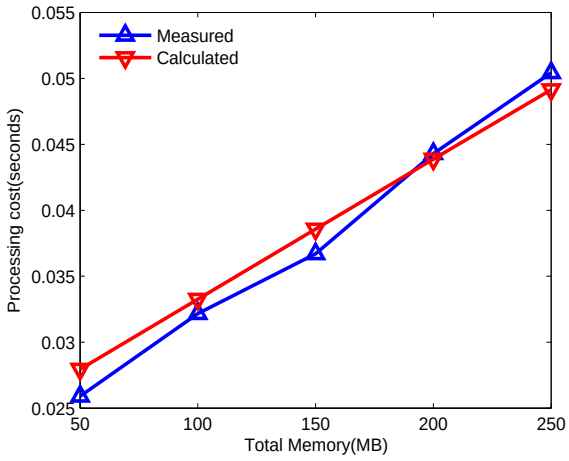


Figure 6.9: Cost validation

6.5.1 Cost Validation

The cost model for X-HYBRIDJOIN has been validated by comparing the calculated cost with the measured cost. Figure 6.9 presents the comparisons of both costs. The figure shows that the calculated cost closely resembles the measured cost, which proves the correctness of the cost model.

6.6 Summary

In this chapter, HYBRIDJOIN has been explored and an extended version of it named X-HYBRIDJOIN has been presented. The main objective in X-HYBRIDJOIN is to minimize the disk access cost by taking into account the real world characteristics of stream data. The cost model for X-HYBRIDJOIN has also been presented. To validate the argument a prototype of X-HYBRIDJOIN has been implemented that demonstrates a significant improvement in performance. In addition to that X-HYBRIDJOIN has been tuned to achieve the optimal memory distribution among the components. The cost model for X-HYBRIDJOIN has also been revised to implement the tuning module. At the end of the chapter it has been proven that X-HYBRIDJOIN with tuning performs significantly better than X-HYBRIDJOIN without tuning.

7

Optimised X-HYBRIDJOIN

7.1 Introduction

This chapter describes an optimisation of X-HYBRIDJOIN. X-HYBRIDJOIN is particularly designed for non-uniform distributions by incorporating Zipf's law into its implementation. According to long-tail market economics, a significant portion of sales comes through a small number of products [3]. Therefore, a small number of pages in the master data are used frequently during the join operation. X-HYBRIDJOIN divides the disk buffer, used to load master data into memory, into two parts. One part is non-swappable. This holds the most frequently used page(s) of master data permanently in memory. The other part is swappable and it exchanges its contents on each iteration of the algorithm. The main argument reflected in the algorithm is that, storing the most frequently-used part of the master data permanently in memory minimizes the disk I/O cost and that eventually amortises the fast incoming stream of updates.

X-HYBRIDJOIN achieves better performance compared to earlier algorithms by pin-

ning frequently-accessed data from the master data in the main memory. Apart from being held in the main memory, X-HYBRIDJOIN does not treat this frequently-accessed data differently from other data coming from the master data. Therefore, the swappable and non-swappbale parts of the disk buffer cannot work independently of each other. This generates some unnecessary processing costs that negatively affect the performance of the algorithm. For example, in each iteration the algorithm matches all the tuples in the non-swappable part of the disk buffer with the hash table, regardless of whether the matching is successful or unsuccessful. This increases the unnecessary look-up cost for the algorithm. Similarly, the algorithm stores all the stream tuples in memory, whether they join with the swappable or the non-swappable part of the disk buffer, increasing the cost in terms of loading and unloading the stream tuples into memory. In contrast if only those stream tuples which join with the swappable part of the disk buffer are stored in memory, it can save the unnecessary costs of loading and unloading those stream tuples which join with non-swappable part of the disk buffer. Also more stream tuples can be accommodated in memory at the same time.

Based on these motivations an improved version of X-HYBRIDJOIN known as Optimised X-HYBRIDJOIN (Optimised Extended Hybrid Join) [85][1] is presented in this chapter. Optimised X-HYBRIDJOIN divides the algorithm into two phases which can work independently. One phase deals with the swappable part while the other phase deals with the non-swappable part of the master data, using appropriate data structures for each phase. In the proposed algorithm, due to choosing an appropriate architecture, all unnecessary costs are minimized and performance is improved significantly. The tuning module for the algorithm is also presented, which is based on a mathematical cost model and which makes the algorithm more efficient.

The remainder of the chapter is structured as follows. Section 7.2 presents Optimised X-HYBRIDJOIN with its execution architecture and pseudo-code. In Section 7.3 the cost model for Optimised X-HYBRIDJOIN is derived. Section 7.4 describes the tuning phase for the algorithm. The experimental study is presented in Section 7.5 and finally Section 7.6 summarises the chapter.

[1]This work has been published in 23rd Australasian Database Conference (ADC'12).

7.2 Optimised X-HYBRIDJOIN

To overcome the problems stated in the introduction section an alternative algorithm called Optimised X-HYBRIDJOIN is proposed. Optimised X-HYBRIDJOIN decomposes the algorithm into two hash join phases that can execute separately. One phase uses R as the probe input; the largest part of R will be stored in tertiary memory. This phase is called disk-probing phase. The other join phase uses the stream as the probe input and it is called stream-probing phase. This phase deals only with a small part of R. For each incoming stream tuple, Optimised X-HYBRIDJOIN first uses the stream-probing phase to find a match for frequent requests quickly, and if no match is found, the stream tuple is forwarded to the disk-probing phase. The details of the proposed algorithm are presented in the following subsections.

7.2.1 Memory Architecture

This section gives a high-level description of Optimised X-HYBRIDJOIN, while a detailed walk-through of the algorithm can be found in Section 7.2.2. From the architectural point of view, the key concept in Optimised X-HYBRIDJOIN is to execute both the disk-probing phase and the stream-probing phase independently, using appropriate data structures. The reason for doing this is to eliminate unnecessary costs, as it is described later in this section.

The memory architecture for Optimised X-HYBRIDJOIN is shown in Figure 7.1. The largest components of Optimised X-HYBRIDJOIN with respect to memory size are two hash tables, one storing stream tuples, denoted by H_S, and the other storing tuples from the master data, denoted by H_R. The other main components of Optimised X-HYBRIDJOIN are a disk buffer, a queue and a stream buffer. R and stream S are the external input sources. Similar to X-HYBRIDJOIN R is assumed to be sorted according to the frequency of access. The hash table H_R contains the most frequently-accessed part of R, which is stored permanently in memory.

Optimised X-HYBRIDJOIN alternates between the stream-probing and disk-probing phases. The hash table H_S is used to store only that part of the update stream which does not match tuples in H_R. A stream-probing phase ends if H_S is completely filled or if the stream buffer is empty. Then the disk-probing phase becomes active. The length of

the disk-probing phase is determined by the fact that only a small number of disk pages of R have to be loaded at one time in order to amortise the costly disk access. In the disk-probing phase of Optimised X-HYBRIDJOIN, the oldest tuple in the queue is used to determine the partition of R that is loaded for a single probe step. This is also the step where Optimised X-HYBRIDJOIN needs an index on table R in order to find the partition in R that matches the oldest stream tuple. After one probe step, a sufficient number of stream tuples are deleted from H_S, so the algorithm switches back to the stream-probing phase. One phase of stream-probing with a subsequent phase of disk-probing constitutes one outer iteration of Optimised X-HYBRIDJOIN. The disk-probing phase could work on its own, without the stream-probing phase. Therefore, the stream-probing phase can be switched-off if it is required and the memory needed for that phase would be reassigned. In this case the algorithm simply operates as HYBRIDJOIN, described in Chapter 5. The stream-probing phase is used to boost the performance of the algorithm by quickly matching the most frequently-used master data. The disk buffer stores the swappable part of R and for each iteration it loads a particular partition of R into the memory. The other component queue is based on a doubly-linked-list, and is used to store the values for the join attribute. Each node in the queue also contains the addresses of its neighbour nodes. The reason for choosing this data structure is to allow random deletion from the queue. The stream buffer is included in the diagram for completeness, but is in reality always a tiny component and will not be considered in the cost model. There are two key advantages of Optimised X- HYBRIDJOIN over X-HYBRIDJOIN. First, due to the independent processing of each phase the stream tuples can be looked-up directly in H_R without loading them into memory. This not only eliminates an unnecessary look-up cost, but also allows more of the stream to be accommodated in memory. In contrast to this, X-HYBRIDJOIN stores a major part of the stream, related to the non-swappable part, in memory and for each iteration, the algorithm looks-up all the tuples of the non-swappable part in the hash table one-by-one. In the situation when the tuples do not match, the algorithm faces an additional look-up cost. Secondly, since Optimised X- HYBRIDJOIN does not store a large part of the stream in memory, it eliminates the costs of loading and unloading that part of the stream into the hash table, H_S. These additional features in Optimised X-HYBRIDJOIN help in reducing the overall processing cost for the algorithm and that eventually improves the performance.

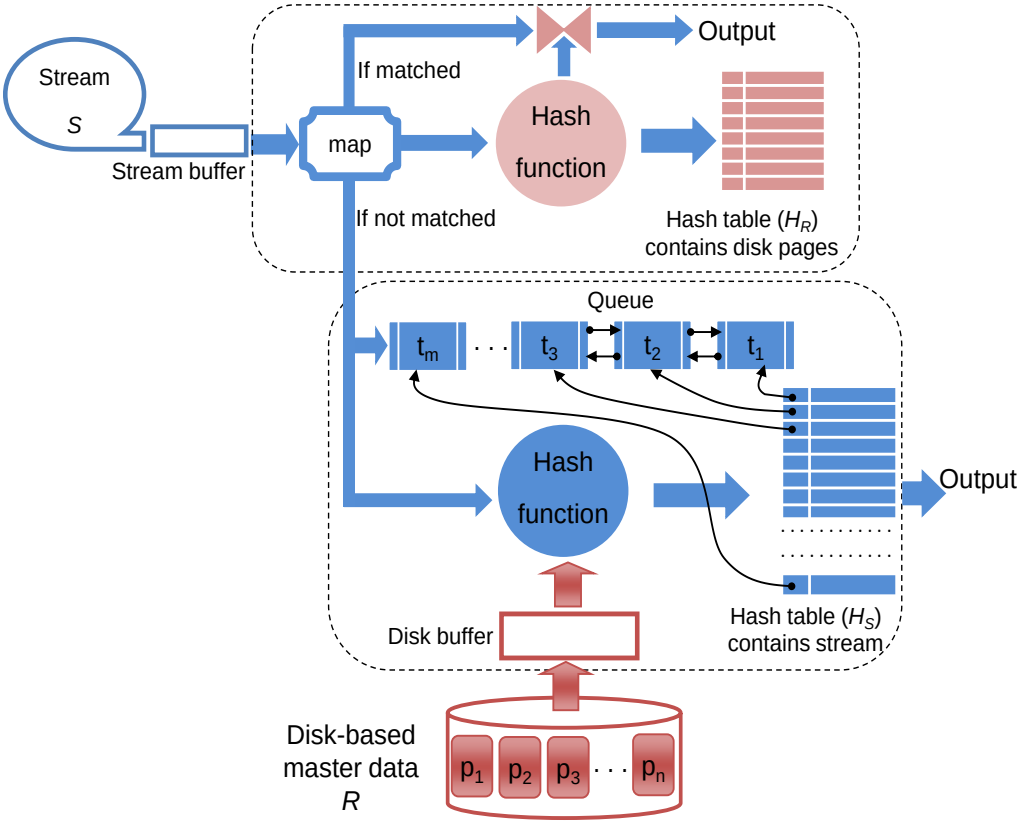


Figure 7.1: Memory architecture for Optimised X-HYBRIDJOIN

7.2.2 Algorithm

After dividing the available memory among the join components, the algorithm starts its execution. The pseudo-code for Optimised X-HYBRIDJOIN is shown in Algorithm 5. The outer loop of the algorithm is an endless loop (line 2). The body of the outer loop has two main phases, the stream-probing phase and the disk-probing phase. Due to the endless loop, these two phases are executed alternately.

Lines 3 to 11 comprise the stream-probing phase. The stream-probing phase has to know the number of empty slots in H_S. This number is kept in variable $hSavailable$. At the start of the algorithm, all the slots in H_S are empty (line 1). The stream-probing phase has an inner loop that continues while stream tuples as well as empty slots in H_S are available (line 3). In the loop, the algorithm reads one input stream tuple t at a time (line 4). The algorithm looks up t in H_R (line 5). In the case of a match, the algorithm generates the join output without storing t in H_S (line 6). In the case where t does not match, the algorithm loads t into H_S, along with enqueuing its key attribute value in the queue (line 8). The counter of empty slots in H_S then has to be decreased (line 9).

Algorithm 5 Optimised X-HYBRIDJOIN

Input: A master data R with an index on join attribute and a stream of updates S.
Output: $R \bowtie S$
Parameters: w (where $w{=}w_S{+}w_N$)tuples of S and k pages of R.
Method:

```
1:  hSavailable ← h_S
2:  while (true) do
3:    while (stream available AND hSavailable > 0) do
4:      READ a stream tuple t from the stream buffer
5:      if t ∈ H_R then
6:        OUTPUT t ⋈ H_R
7:      else
8:        ADD the stream tuple t into H_S while also placing its join attribute values
          into Q
9:        hSavailable ← hSavailable − 1
10:     end if
11:   end while
12:   READ the oldest join attribute value from Q
13:   READ a partition of R into the disk buffer using the oldest join attribute value
      from the queue for the index look-up.
14:   for each tuple r in the disk buffer do
15:     if r ∈ H_S then
16:       OUTPUT r ⋈ H_S
17:       f ← number of matching tuples found in H_S
18:       DELETE all matched tuples from H_S along with the corresponding nodes from
          Q
19:       hSavailable ← hSavailable + f
20:     end if
21:   end for
22: end while
```

Lines 12 to 21 comprise the disk-probing phase. At the start of this phase, the algorithm reads the oldest key attribute value from the queue and loads a partition of R into the disk buffer, using that key attribute value as an index (lines 12 and 13). In an inner loop, the algorithm looks up all tuples r from the disk buffer in hash table H_S one-by-one. In the case of a match, the algorithm generates the join output (line 16). Since H_S is a multi-hash-map, there can be more than one match, the number of matches being f (line 17). The algorithm removes all matching tuples from H_S along with deleting the corresponding nodes from the queue (line 18). This creates empty slots in H_S (line 19). In the next outer iteration the algorithm fills these empty slots if stream input is available.

Table 7.1: Some new symbols used in Optimised X-HYBRIDJOIN

Parameter name	Symbol
Number of stream tuples processed in each iteration through H_R	w_N
Number of stream tuples processed in each iteration through H_S	w_S
Size of disk buffer (pages)	k
Size of disk buffer (tuples)	$d = k\frac{v_P}{v_R}$
Size of H_R (pages)	l
Size of H_R (tuples)	$h_R = l\frac{v_P}{v_R}$
Size of H_S (tuples)	h_S

7.3 Cost Model

As the execution layout of Optimised X-HYBRIDJOIN is different from that of simple X-HYBRIDJOIN, this needs to revise the cost model for Optimised X-HYBRIDJOIN. The main objective in developing the cost model is to inter-relate the key parameters like the algorithm input size w, the processing cost c_{loop} for these w tuples, the available memory M and the service rate μ. The other important application for the cost model is in the tuning process, where the optimal size is determined for each component of the algorithm. The details of the tuning process are presented in Section 7.4. As stated in earlier chapters, the main costs for an algorithm are described in terms of memory and processing time. The memory cost in the case of Optimised X-HYBRIDJOIN is exactly the same as for X-HYBRIDJOIN, due to their having the same number of components. However, the processing cost is different due to the different execution layouts. Most of the notations used in this cost model have already been specified in Table 3.1 and Table 4.1 however some further symbols are shown in Table 7.1.

7.3.1 Memory Cost

As described above, the memory cost for Optimised X-HYBRIDJOIN is the same as that for X-HYBRIDJOIN. Therefore, Equation 6.4 (which is 7.1 here) describes the memory cost for Optimised X-HYBRIDJOIN.

$$M = k \cdot v_P + l \cdot v_P + \alpha\left[M - (k+l)v_P\right] + (1-\alpha)\left[M - (k+l)v_P\right] \tag{7.1}$$

7.3.2 Processing Cost

Due to its different execution layout, the processing cost of Optimised X-HYBRIDJOIN is different from that of X-HYBRIDJOIN and, therefore, it is calculated here. To make it simpler the processing cost for individual components is calculated first and then all these costs are summed up to calculate the total processing cost for one iteration.
Cost to read the most frequent l number of pages of R into H_R= $c_{I/O}(l \cdot v_P)$
Cost to read k number of pages into the disk buffer= $c_{I/O}(k \cdot v_P)$
Cost to look-up w_N tuples into $H_R = w_N \cdot c_H$
Cost to look-up disk buffer tuples into H_S= $d \cdot c_H$
Cost to generate the output for w_N tuples= $w_N \cdot c_O$
Cost to generate the output for w_S tuples= $w_S \cdot c_O$
Cost to read w_N tuples from the stream buffer= $w_N \cdot c_S$
Cost to read w_S tuples from the stream buffer= $w_S \cdot c_S$
Cost to append w_S tuples into the H_S and the queue= $w_S \cdot c_A$
Cost to delete w_S tuples from the H_S and the queue= $w_S \cdot c_E$
The hash table H_R is filled only once before the actual execution of the algorithm starts; therefore its cost is excluded. By aggregating the costs for individual components, the total cost for one loop iteration is:

$$c_{loop} = 10^{-9} \left[c_{I/O}(k.v_P) + d \cdot c_H + w_S(c_O + c_E + c_S + c_A) + w_N(c_H + c_O + c_S) \right] \quad (7.2)$$

Since the algorithm processes w_N and w_S tuples of stream S in c_{loop} seconds, the service rate μ can be calculated using Equation 7.3.

$$\mu = \frac{w_N + w_S}{c_{loop}} \quad (7.3)$$

7.4 Tuning

Normally the stream-based join algorithms are executed online, where limited memory resources are available. Due to the fixed and small amount of available memory, each component in the join faces a trade-off with respect to memory distribution. Assigning more memory to one component means assigning less memory to some other components.

On close observation it can be seen that a component like the hash table H_S, used to store the stream, requires more memory compared to the other components, such as the disk buffer, the stream buffer and the hash table H_R, used to store the frequently-used disk pages. The disk buffer and the hash table H_R are the key components for tuning, and the memory assigned to the other components depends on them. The reason for tuning the disk buffer is that the dominant I/O cost is directly connected to the disk buffer. The tuning of the algorithm uses the cost model that has been derived before. Tuning is not performed merely using a theoretical approach, rather the optimal tuning settings are approximated using an empirical approach. Finally the experimentally-obtained tuning results are compared with the results obtained using the cost model.

7.4.1 Tuning using Empirical Approach

This section focuses on the tuning of key components, namely the disk buffer and the hash table H_R using an empirical approach. The performance of the algorithm has been tested for a set of values for both components, rather than for every consecutive value. It has been assumed that the total allocated memory and the size of the master data are fixed. The sizes for the disk buffer and the hash table H_R are varied in such a way that for each size of the disk buffer the performance is measured against a series of values for the size of H_R. The performance measurements for the grid of values for the sizes of disk buffer denoted by d and the size of H_R denoted by h_R are shown in Figure 7.2. The figure shows that, if the performance for each fixed value of d is observed against all values of h_R, in the beginning the performance increases rapidly with an increase in h_R. However, after reaching a particular value of h_R, the performance starts decreasing with further increases in h_R. A plausible reason for this behavior is that initially, increasing h_R increases the probability of matching the stream tuples with H_R rapidly. After attaining the optimal value, further incrementing h_R makes no significant difference to the stream-matching probability, due to the skew factor in stream distribution. On the other hand, the associated reduction in memory size for the hash table H_S means that the performance begins to decrease. Similarly when the performance is analysed for each fixed value of h_R against all the values of d, initially the performance increases, since the costly disk access is amortised for a larger number of stream tuples. After attaining a maximum, the

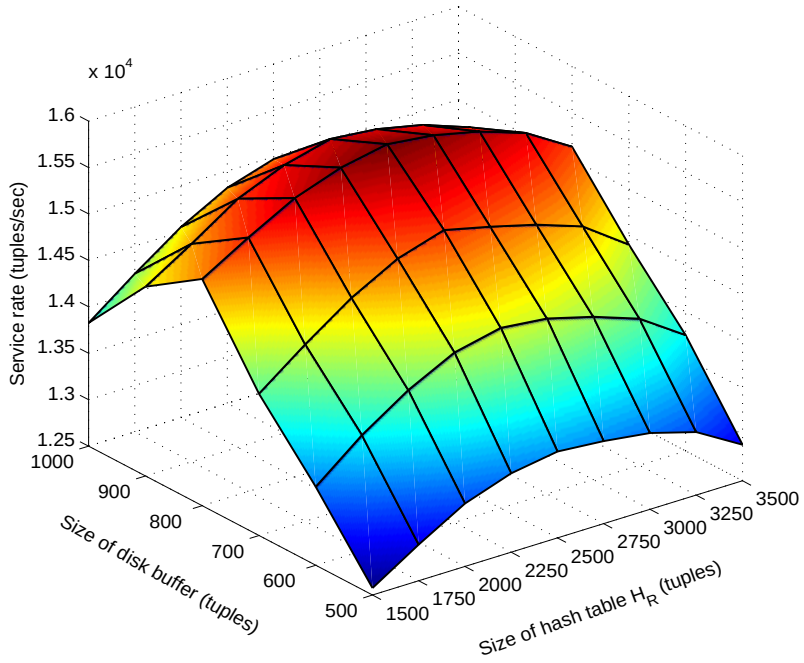


Figure 7.2: Tuning of Optimised X-HYBRIDJOIN using empirical approach

performance decreases because of the increase in I/O cost for loading more of R at one time in a non-selective way.

The figure shows that the optimal memory settings for both the disk buffer and the hash table H_R can be determined by considering the intersection of the values of both components at which the algorithm individually performs at a maximum.

7.4.2 Tuning based on Cost Model

To validate the cost model against measurements, the algorithm has also been tuned based on the cost model. Under Equation 7.3 the service rate depends on the values of w_S, w_N and the cost c_{loop}. Therefore, to determine the settings at which the algorithm performs optimally, it is first necessary to calculate the sizes of w_N and w_S. A similar approach for mathematical tuning to that described in Chapter 6 is used here. However, the formulas to calculate w_N and w_S are different from the formula to calculate w in Chapter 6. The reason is that in Optimised X-HYBRIDJOIN the processing of the stream-probing phase is independent of the disk-probing phase. Also the stream related to the stream-probing phase is not stored in memory. Therefore, most cost factors related to the stream-probing phase are removed. In contrast to X-HYBRIDJOIN, w_S and w_N need to be calculated separately.

Mathematical model to calculate w_N: The main components that directly affect w_N are the total size of R (denoted by R_t) on the disk and the size of the hash table H_R (denoted by h_R) that contains the most frequently-used part of R in the memory. If the stream of updates S is formulated using Zipfs law with the exponent value being equal to 1, then the matching probability p_N for stream S with H_R can be determined using Equation 7.4.

$$p_N = \frac{\sum_{x=1}^{h_R} \frac{1}{x}}{\sum_{x=1}^{R_t} \frac{1}{x}} \approx \frac{\ln(h_R)}{\ln(R_t)} \tag{7.4}$$

Now using Equation 7.4 the constant factors of change can be determined in p_N by changing the values of h_R and R_t individually. Assumes that p_N decreases by a constant factor ϕ_N if the value of R_t is doubled, and increases by a constant factor ψ_N if the value of h_R is doubled. Knowing these constant factors the value of w_N can be calculated. Consider a hypothesis

$$p_N = R_t^y h_R^z \tag{7.5}$$

where y and z are unknown constants whose values need to be determined.

By doubling R_t, the matching probability p_N decreases by a constant factor ϕ_N, Equation 7.5 becomes:

$$\phi_N p_N = (2R_t)^y h_R^z$$

Dividing the above equation by Equation 7.5 we get $2^y = \phi_N$ and therefore, $y = \log_2(\phi_N)$. Similarly by doubling h_R the matching probability p_N increases by a constant factor ψ_N therefore, Equation 7.5 can be written as:

$$\psi_N p_N = R_t^y (2h_R)^z$$

By dividing the above equation by Equation 7.5 we get $2^z = \psi_N$ and therefore, $z = \log_2(\psi_N)$. After putting the values of constants y and z in Equation 7.5 we get:

$$p_N = R_t^{\log_2(\phi_N)} h_R^{\log_2(\psi_N)}$$

If S is the total number of stream tuples that are processed (through both the stream-probing and disk-probing phases) in N iterations, then w_N can be calculated using Equa-

tion 7.6

$$w_N = \frac{(R_t^{\log_2(\phi_N)} h_R^{\log_2(\psi_N)})S}{N} \tag{7.6}$$

Mathematical model to calculate w_S: The second phase of the Optimised X-HYBRIDJOIN algorithm, also called the disk-probing phase, deals with the rest of the master data R' (where $R' = R_t - h_R$), which occurs less frequently in the stream input as compared to that part which exists permanently in memory. The algorithm reads R' in partitions while the size of each partition is equal to the size of the disk buffer d. As mentioned earlier, the daily market transactions typically formulate the Zipfian distribution, which means that matching probability for every next partition in R' is less than the previous one. Therefore, the matching probability for each partition is calculated by taking the summation over the discrete Zipfian distribution separately and then aggregating all of them as shown below.

$$\sum_{x=h_R+1}^{h_R+d} \frac{1}{x} + \sum_{x=h_R+d+1}^{h_R+2d} \frac{1}{x} + \sum_{x=h_R+2d+1}^{h_R+3d} \frac{1}{x} + \cdots + \sum_{x=h_R+(n-1)d+1}^{h_R+nd} \frac{1}{x}$$

We simplify this to:

$$\sum_{x=h_R+1}^{h_R+nd} \frac{1}{x} \Rightarrow \sum_{x=h_R+1}^{R_t} \frac{1}{x}$$

From this the average matching probability $\overline{p}_S$ can be obtained in the disk probe phase, which is needed for calculating w_S. Let n be the total number of partitions in R', then the average matching probability $\overline{p}_S$ can be determined by dividing the above summation by n. In the denominator, a similar normalization term to that used in Equation 7.4 is used.

$$\overline{p}_S = \frac{\sum_{x=h_R+1}^{R_t} \frac{1}{x}}{n \sum_{x=1}^{R_t} \frac{1}{x}} = \frac{\ln(R_t) - \ln(h_R + 1)}{n(\ln(R_t) + \gamma)} \tag{7.7}$$

To determine the effects of d, h_R and R_t on $\overline{p}_S$ the same number of steps is required as in the case of w_N. If d is doubled then n will be halved in Equation 7.7 and therefore, the value of $\overline{p}_S$ increases with a constant factor of θ_S. Similarly, if h_R and R_t are doubled one-by-one in Equation 7.7, the value of $\overline{p}_S$ decreases with a constant factor of ψ_S and ϕ_S

respectively. A similar hypothesis is considered here as in Equation 7.5.

$$\overline{p}_S = d^x h_R^y R_t^z \tag{7.8}$$

The values for the constants x, y and z in this case will be $x = \log_2(\theta_S)$, $y = \log_2(\psi_S)$ and $z = \log_2(\phi_S)$ respectively. Therefore by replacing the parameters with constants, Equation 7.8 will become.

$$\overline{p}_S = d^{\log_2(\theta_S)} h_R^{\log_2(\psi_S)} R_t^{\log_2(\phi_S)}$$

If h_S are the number of stream tuples stored in the hash table then the average value for w_S can be calculated using Equation 7.9.

$$w_S(\text{average}) = d^{\log_2(\theta_S)} h_R^{\log_2(\psi_S)} R_t^{\log_2(\phi_S)} h_S \tag{7.9}$$

Once the values of w_N and w_S have been determined, the algorithm can be tuned using Equation 7.3.

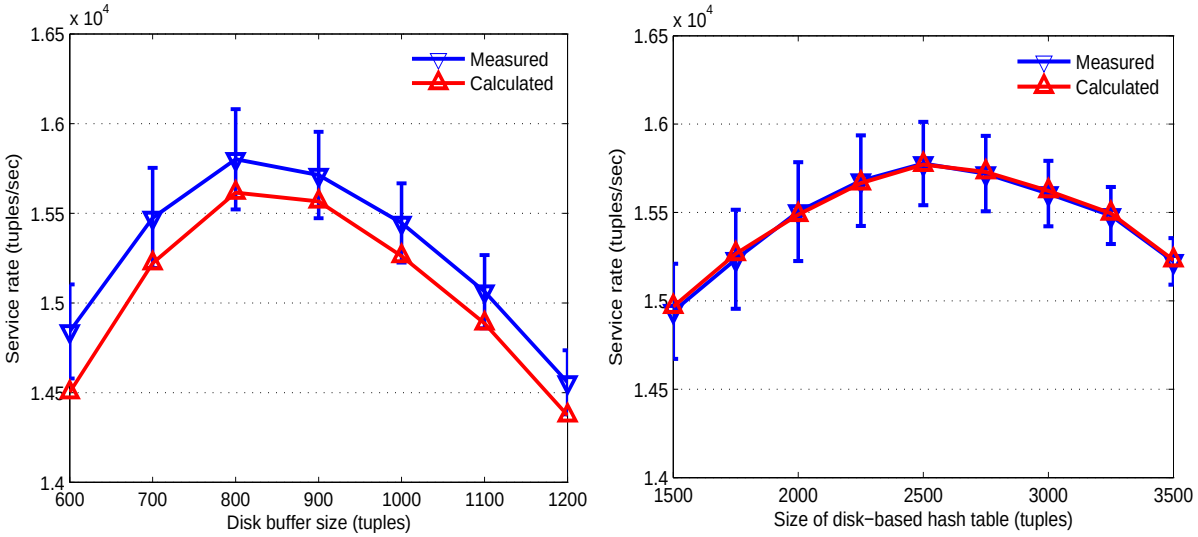
A number of experiments have been conducted to observe the effect of all necessary components on w_N and w_S. The results and detailed descriptions of these experiments can be found in Appendix C.

7.4.3 Comparisons of both Tuning Approaches

In this section, in order to validate the cost model, the tuning results obtained through measurements are compared with the tuning results calculated using the cost model.

Tuning of the swappable part: In this experiment tuning of the disk buffer is performed using both the empirical and the mathematical approaches. The tuning results of each approach are shown in Figure 7.3(a). It can be observed that the results in both cases are very similar, with a deviation of only 1.5%.

Tuning of the non-swappable part: Tuning comparisons are also made for the hash table H_R using both approaches. The experimental results in this case are shown in Figure 7.3(b). The results in both cases are again closely related, with a deviation of only 0.33%. This proves the accuracy of the cost model.



(a) Tuning Comparison for swappable part: based on measurements vs cost model

(b) Tuning Comparison for non-swappable part: based on measurements vs cost model

Figure 7.3: Tuning comparisons for Optimised X-HYBRIDJOIN using both empirical and mathematical approaches

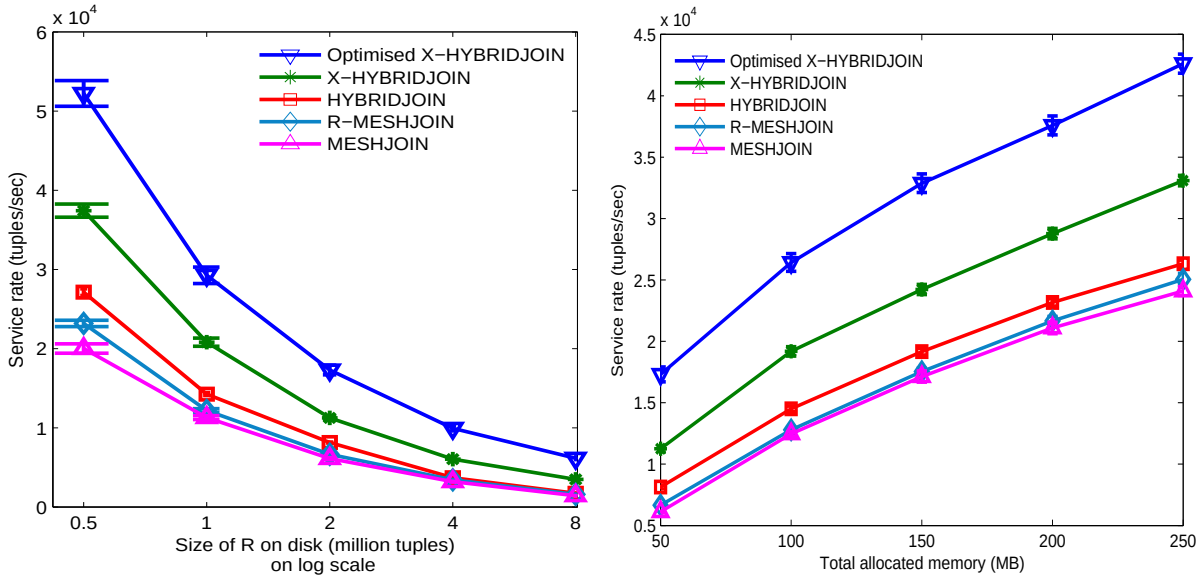
7.5 Experimental Study

This section presents a series of experimental results. The experiments are conducted in two dimensions. Section 7.5.1 compares the performance of Optimised X-HYBRIDJOIN with other related algorithms. While Section 7.5.2 compares the calculated cost for Optimised X-HYBRIDJOIN with the measured cost.

7.5.1 Performance Evaluation

Similar to the performance analysis strategy used in previous chapters, the performance of Optimised X-HYBRIDJOIN is compared by varying two parameters, the total allocated memory M and the size of the master data R.

Performance comparisons when the size of R varies: In this experiment the performance of Optimised X-HYBRIDJOIN has been compared with other join algorithms. In this experiment it has been assumed that the size of the master data R varies exponentially while the total allocated memory is fixed for all values of R. The performance results are shown in Figure 7.4(a). It is clear that for all settings of R the performance in the case of Optimised X-HYBRIDJOIN is significantly better than that of the other algorithms.



(a) Performance comparison with 95% confidence interval while M= *50MB* and R_t varies.

(b) Performance comparison with 95% confidence interval while R_t=*2* million tuples and M varies.

Figure 7.4: Performance comparisons of Optimised X-HYBRIDJOIN with other join algorithms

Performance comparisons for different memory budgets: In the second experiment the performance of all algorithms has been tested using different memory budgets while keeping the size of R fixed (2 million tuples). Figure 7.4(b) presents the comparisons between all the approaches. For all memory budgets, Optimised X-HYBRIDJOIN again performs significantly better than the other approaches.

In both scenarios the reason for the improvement in performance is the better execution layout of Optimised X-HYBRIDJOIN. In X-HYBRIDJOIN the data structures used for some components are ineffective, causing unnecessary costs in processing the stream tuples, and eventually this affects the performance of the algorithm negatively.

7.5.2 Cost Validation

In the second part of the experiments the cost model for each algorithm is validated by comparing the calculated cost with the measured cost. Figure 7.5 presents the comparisons of these costs. From the figure it can be observed that for each memory setting the calculated cost is closely matched with the measured cost.

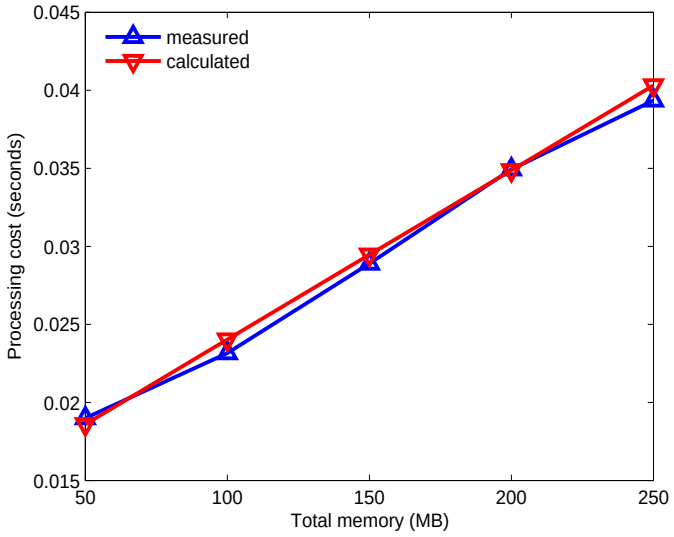


Figure 7.5: Cost validation

7.6 Summary

In this chapter a significant optimisation has been presented for X-HYBRIDJOIN. X-HYBRIDJOIN was designed to make use of non-uniformly distributed data as found in real-world applications. In the investigation it has been discovered that the algorithm has some architectural limitations which affect its performance. Data structures used for some components, such as the non-swappable part of the disk buffer, are not optimal, and create an additional look-up cost. In addition, the algorithm stores a major part of the stream in memory. That part of the stream matches with the non-swappable part of the disk buffer. This is unnecessary and generates extra costs in the form of loading and unloading stream tuples into memory. On the basis of these observations an optimised version of the existing X-HYBRIDJOIN called Optimised X-HYBRIDJOIN has been presented. In Optimised X-HYBRIDJOIN, efficient data structures allow stream tuples that match with the non-swappable part of the master data to be processed independently from those that match with the swappable part of the master data. The stream that matches with the non-swappable part does not need to be stored in memory. This has two advantages: (a) it eliminates the additional costs required for loading and unloading the stream tuples into memory; (b) more stream tuples related to the swappable part can be accommodated in memory. Mathematical costs have been calculated for the algorithm. To obtain the maximum performance the algorithm has been tuned empirically as well as mathematically (based on the cost model). A prototype of Optimised X-HYBRIDJOIN

has been implemented to compare its performance with that of related algorithms. The experiments have proved that Optimised X-HYBRIDJOIN performs significantly better than the other related approaches.

8

Generalisation of Optimised X-HYBRIDJOIN

8.1 Introduction

Chapter 7 presented a stream-based algorithm called Optimised X-HYBRIDJOIN, which is designed particularly for non-uniform distributions. The algorithm consists of two major phases, the stream-probing phase and the disk-probing phase, both of which can execute independently using efficient data structures. The experiments presented in Chapter 7 show that Optimised X-HYBRIDJOIN produces a better service rate than the other approaches described in this research. However, the algorithm assumes that master data is sorted on the basis of the frequency with which it has been accessed. Therefore, when the most frequently accessed tuples in the master data change while the algorithm is running, the performance of the algorithm slows down due to the presence of unsorted master data. To scale up the performance, the master data needs to be sorted again and

this interrupts the working of the algorithm.

This issue provides motivation for presenting a generalised form of Optimised X-HYBRIDJOIN called CACHEJOIN (Cache Join). CACHEJOIN introduces a new operation known as frequency detection. The main purpose of this operation is to calculate the frequency of matching for each disk tuple when it is loaded in the disk-probing phase. If the frequency for any disk tuple crosses a certain threshold limit, that disk tuple is switched into the stream-probing phase. The stream-probing phase contains those disk tuples which are frequent in the input stream. Further details about CACHEJOIN are presented in Section 8.2.

The key benefit of CACHEJOIN is that it removes the constraint of sorting the master data. The experiments presented at the end of the chapter show that, for non-uniform distributions, which are common in real-world applications [32, 63, 65, 67, 74, 89, 98, 106, 115, 116, 117], CACHEJOIN performs significantly better than MESHJOIN, R-MESHJOIN and HYBRIDJOIN. However, CACHEJOIN performs slightly worse than Optimised X-HYBRIDJOIN due to the additional costs required to compare the frequency of matching for each disk tuple with the given threshold value and to switch the disk tuple into the stream-probing phase if the condition is true. But against this small performance loss CACHEJOIN removes the constraint of sorting the master data, which is an acceptable contribution.

The rest of the chapter is structured as follows. Section 8.2 presents the memory architecture, the execution layout and the pseudo-code for CACHEJOIN. In Section 8.3 the cost model for CACHEJOIN is derived. Section 8.4 describes the tuning module for the algorithm. The experimental study is presented in Section 8.5 and finally Section 8.6 summarises the chapter.

8.2 CACHEJOIN

The main objective of CACHEJOIN is to eliminate the constraint of sorting the master data with respect to the access frequency. CACHEJOIN performs optimally for non-uniform distributions while unsorted master data exists on disk.

8.2.1 Data Structures and Execution Architecture

This section gives a high-level description of CACHEJOIN, the detailed execution steps of the algorithm being described in Section 8.2.2. Similar to Optimised X-HYBRIDJOIN the CACHEJOIN algorithm also possesses two complementary hash join phases, somewhat similar to Symmetric Hash Join [108, 109]. The memory architecture and the working layout of the CACHEJOIN algorithm are mostly identical to the Optimised X-HYBRIDJOIN algorithm except for the addition of one new operation, frequency detection. This section does not repeat the details about the operations which are common between CACHEJOIN and Optimised X-HYBRIDJOIN, and focuses only on the newly-added operation.

The execution architecture for CACHEJOIN is shown in Figure 8.1. As described in the beginning of this section, the CACHEJOIN algorithm deals effectively with non-uniform stream input by caching the frequent tuples of master data on a permanent basis. In addition, the algorithm eliminates the constraints of sorting the master data. In order to achieve these objectives, the algorithm compares the total number of matches against each disk tuple with a certain threshold value while executing the disk probing phase. If the total number of matches for any disk tuple is greater than the threshold value, that tuple is then switched into the hash table H_R. Moreover, the algorithm uses multi-hash-map for storing the stream tuples in memory storing multiple stream tuples with the same join attribute value. One advantage of choosing this data structure is that the algorithm does not need an additional operation to determine the frequency of matching each disk tuple. Further details about the process are presented in Section 8.2.3. The stream-probing phase is used to boost the performance of the algorithm by quickly matching the frequently-used master data. The question of where to set the threshold arises, i.e. how frequently must a stream tuple be used in order to get into this phase, so that the memory sacrificed for this phase really delivers a performance advantage. In Section 8.3 a precise and comprehensive analysis is given that shows that a remarkably small stream-probing phase can deliver a substantial performance gain. In fact, CACHEJOIN will be tuned to a provably optimal distribution of memory between the two phases, and the components within the phases. In order to corroborate this theoretical model, the experimental performance measurements are also provided later in this chapter to show that the model is highly accurate.

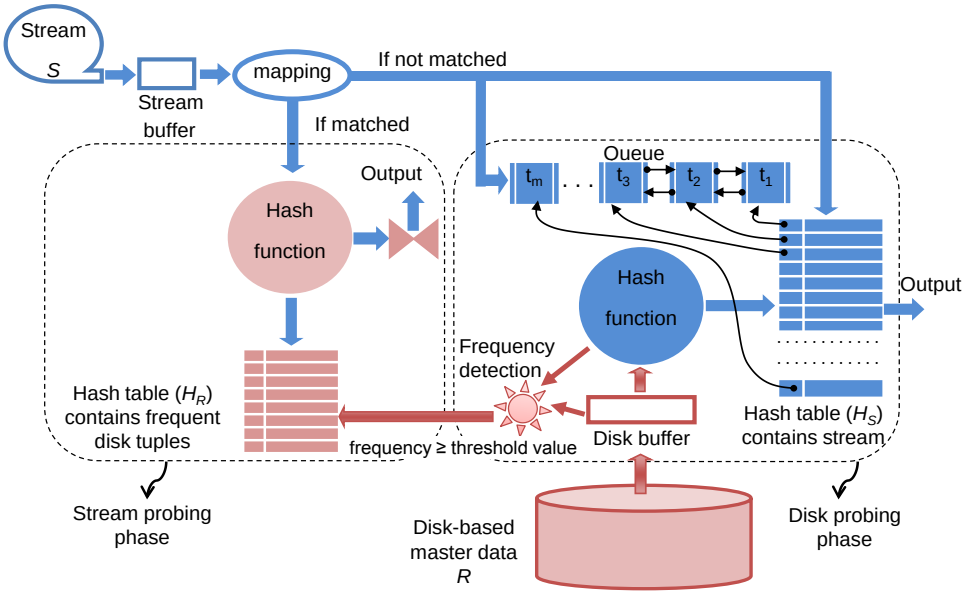


Figure 8.1: Data structure and architecture of CACHEJOIN

One additional feature that CACHEJOIN supports is the changing of the size of the master data on disk while the algorithm is running. It does not interrupt the algorithm, unlike the some existing approaches e.g. X-HYBRIDJOIN and Optimised X-HYBRIDJOIN.

8.2.2 Algorithm

The pseudo-code for CACHEJOIN, shown in Algorithm 6, is mostly similar to that of Optimised X-HYBRIDJOIN stated in Chapter 7 with only the addition of some steps about frequency comparison. The outer loop of the algorithm is an endless loop (line 2). The body of the outer loop has two main parts, the stream-probing phase and the disk-probing phase. Due to the endless loop, these two phases are executed alternately.

Lines 3 to 11 comprise the stream-probing phase. The stream-probing phase has to know the number of empty slots in H_S. This number is kept in variable $hSavailable$. At the start of the algorithm, all slots in H_S are empty (line 1). The stream-probing phase has an inner loop that continues while stream tuples and empty slots in H_S are available (line 3). In the loop, the algorithm reads one input stream tuple t at a time (line 4). The algorithm looks up t in H_R (line 5). In the case of a match, the algorithm generates the join output without storing t in H_S (line 6). In the case that t does not match, the algorithm loads t into H_S while also enqueuing its key attribute value in the queue (line

8). The counter of empty slots in H_S has to be decreased (line 9).

Lines 12 to 24 comprise the disk-probing phase. At the start of this phase, the algorithm reads the oldest key attribute value from the queue and loads a segment of R into the disk buffer, using that key attribute value as an index (line 12, 13). In an inner loop, the algorithm looks up all tuples r one-by-one from the disk buffer into hash table H_S. In the case of a match, the algorithm generates the join output (line 16). Since H_S is a multi-hash-map, there can be more than one match, the number of matches being f (line 17). The algorithm removes all matching tuples from H_S while also deleting the corresponding nodes from the queue (line 18). This creates empty slots in H_S (line19). Lines 20 to 22 are concerned with the frequency comparison and are explained separately in the next section.

8.2.3 Frequency Comparison

The frequency comparison is described in lines 20 to 22 of the algorithm. Line 20 tests whether the matching frequency f of the current tuple is larger than a pre-set threshold. If yes, then this tuple is entered into H_R. If there is no empty slot in H_R the algorithm overwrites any existing tuple in H_R and increases the threshold value by one. The threshold is a flexible barrier. Initially, an appropriate value is assigned to it while later on this value can be varied depending on the frequency of disk tuples.

8.3 Cost Calculation

In this section the cost model is developed for the proposed CACHEJOIN. The memory cost for CACHEJOIN is the same as that for Optimised X-HYBRIDJOIN because no separate component is required to implement the frequency detection operation. However, the processing cost for CACHEJOIN is slightly different from that of Optimised X-HYBRIDJOIN due the one additional operation. It is intuitively clear that there is a trade-off between the memory consumption of the various components. The equations of the cost model are used in the tuning process to find the optimal size for each component of the algorithm using calculus of variations. The notations used in the cost model are given in Tables 3.1, 4.1 and 7.1.

Algorithm 6 CACHEJOIN

Input: A master data R with an index on join attribute and a stream of updates S.
Output: $R \bowtie S$
Parameters: w (where w=w_S+w_N)tuples of S and k number of pages of R.
Method:

1: $hSavailable \leftarrow h_S$
2: **while** (true) **do**
3: **while** (stream available AND $hSavailable > 0$) **do**
4: READ a stream tuple t from the stream buffer
5: **if** $t \in H_R$ **then**
6: OUTPUT $t \bowtie H_R$
7: **else**
8: ADD stream tuple t into H_S along with placing its join attribute values into Q
9: $hSavailable \leftarrow hSavailable - 1$
10: **end if**
11: **end while**
12: READ the oldest join attribute value from Q
13: READ a segment of R into the disk buffer using the oldest join attribute value from the queue for the index look-up.
14: **for** each tuple r in the disk buffer **do**
15: **if** $r \in H_S$ **then**
16: OUTPUT $r \bowtie H_S$
17: $f \leftarrow$ number of matching tuples found in H_S
18: DELETE all matched tuples from H_S along with the corresponding nodes from Q
19: $hSavailable \leftarrow hSavailable + f$
20: **if** ($f \geq thresholdvalue$) **then**
21: SWITCH the tuple r_i into hash table H_R
22: **end if**
23: **end if**
24: **end for**
25: **end while**

8.3.1 Memory Cost

As mentioned above, the memory cost for CACHEJOIN and Optimised X-HYBRIDJOIN are the same therefore, Equation 7.1 is reused here for calculating the memory cost of the CACHEJOIN components.

8.3.2 Processing Cost

Due to an additional operation the processing cost of CACHEJOIN is slightly different from that of Optimised X-HYBRIDJOIN and therefore, it needs to calculate the process-

ing cost for CACHEJOIN. Similar to the previous strategies the processing cost for each individual component is calculated first and then these costs are accumulated to calculate the total processing cost for one iteration.

I/O cost to load the swappable part (nanosec)= $c_{I/O}(k.v_P)$

Cost to load frequently used tuples of master data into the disk-based hash table(nanosec)= $h_R.c_S$

Cost for probing w_N stream tuples in the disk-based hash table (nanosec)= $w_N.c_H$

Probing cost for the swappable part (nanosec)= $d.c_H$

Cost to compare the frequency of all the tuples in the disk buffer with the threshold value (nanosec)= $d.c_F$

Cost to generate output for w_N tuples (nanosec)= $w_N.c_O$

Cost to generate output for w_S tuples (nanosec)= $w_S.c_O$

Cost to read w_N tuples from stream S (nanosec)= $w_N.c_S$

Cost to read w_S tuples from stream S (nanosec)= $w_S.c_S$

Cost to delete w_S tuples from the hash table and the queue (nanosec)= $w_S.c_E$

Cost to append w_S tuples into the hash table and the queue (nanosec)= $w_S.c_A$

Since the most frequently used disk-based tuples are loaded into the disk-based hash table only once and they are very small in number, $h_R.c_S$ is ignored. By aggregating all the above costs except $h_R.c_S$ the total cost of the algorithm for one iteration can be calculated as given in Equation 8.1.

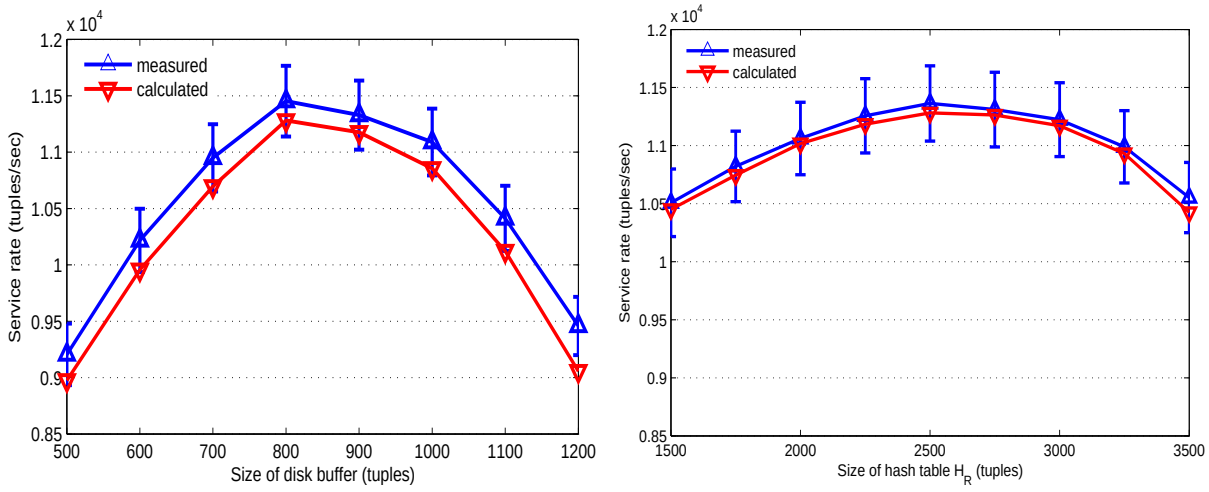
$$c_{loop}(secs) = 10^{-9}[c_{I/O}(k.v_P)+d(c_H+c_F)+w_S(c_O+c_E+c_S+c_A)+w_N(c_H+c_O+c_S)] \quad (8.1)$$

Since the algorithm processes w_N plus w_S tuples of stream S in c_{loop} seconds, the service rate μ can be calculated using Equation 8.2.

$$\mu = \frac{w_N + w_S}{c_{loop}} \quad (8.2)$$

8.4 Tuning

Similar to the other approaches the CACHEJOIN algorithm is also tuned in order to obtain the optimal performance under limited resources. Both empirical and mathe-



(a) Tuning of disk buffer (b) Tuning of hash table H_R containing disk pages

Figure 8.2: Tuning Comparisons: empirical approach vs mathematical approach

matical approaches for tuning the CACHEJOIN algorithm are quite similar to those which have been presented for the Optimised X-HYBRIDJOIN algorithm. The reason for this similarity is that, CACHEJOIN also implements the two phases, stream-probing and disk-probing, in an identical way. However, the tuning results obtained in the case of CACHEJOIN are different from those which have been achieved in Optimised X-HYBRIDJOIN because of the additional cost due to the frequency comparison operation. Therefore, the details of the tuning approaches are skipped and the comparisons of the tuning results using both empirical and mathematical approaches are made directly.

8.4.1 Comparisons of Tuning Results

In this section the tuning results obtained through measurements are compared with the tuning results calculated using the cost model. Figure 8.2(a) shows the empirical and the mathematical tuning results for the disk buffer size d. It can be observed from the figure that the results in both cases are reasonably similar, with a deviation of only 2.5%.
Figure 8.2(b) shows the empirical and the mathematical tuning results for the size of hash table H_R. Again it is fair to say that the results in both cases are reasonably similar, with a deviation of only 0.65%. This is already a corroboration of the accuracy of the cost model.

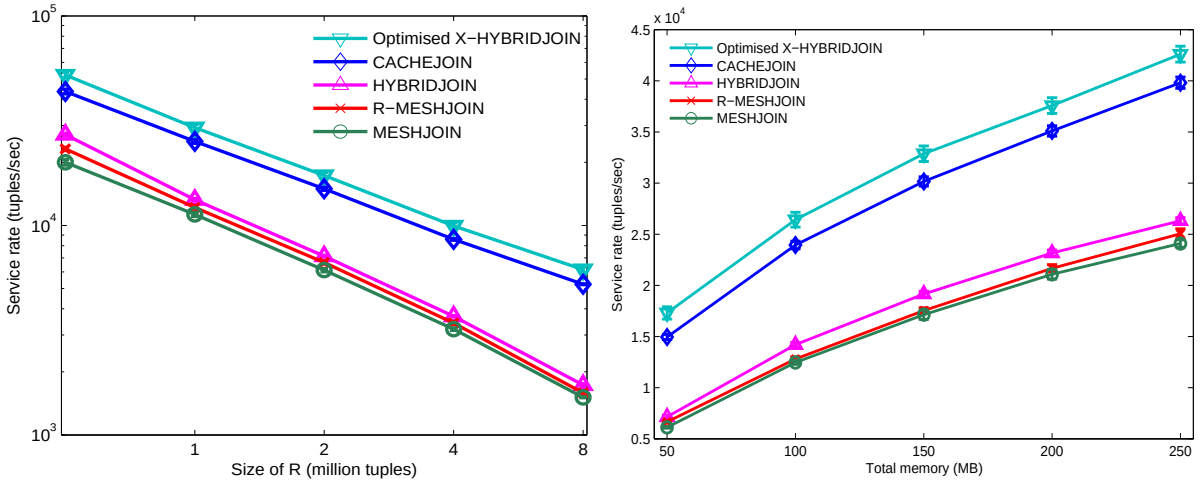
8.5 Performance Experiments

This section presents a series of experimental comparisons between CACHEJOIN and other relevant approaches. The main objective is to analyse how much less efficiently CACHEJOIN performs as compared to Optimised X-HYBRIDJOIN. However, both of these approaches are also compared with HYBRIDJOIN, R-MESHJOIN and MESHJOIN to clarify how much the stream-probing phase contributes in the performance. This is necessary because none of the other three approaches, HYBRIDJOIN, R-MESHJOIN and MESHJOIN, contain the stream-probing phase.

The behaviour of the algorithm has been tested using three different parameters. These three parameters are: the size of the master data table R, the total memory available, and the exponent of the Zipfian distribution. In the presentation here, for the sake of brevity, the discussion for each parameter has been restricted to the one-dimensional variation, i.e. one parameter has been varied at a time.

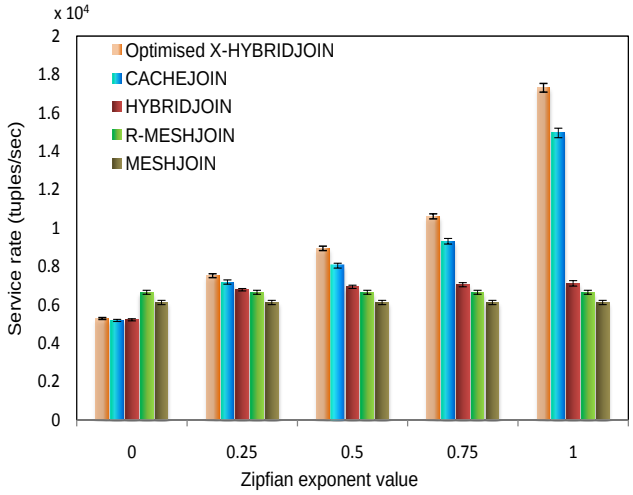
Performance comparisons for varying size of *R*: In this experiment the performance of all approaches has been tested by varying the size of the master data. While the values of the other two parameters are fixed, the Zipfian exponent is equal to 1 and the total memory available is equal to 50MB. The exponent value of the Zipfian distribution has been set to 1 because it is a type of skew that is observed frequently in practice. The discrete sizes of the parameter, the size of the master data R, from a simple geometric progression have been chosen. One thing which is important to mention here is that it is only for Optimised X-HYBRIDJOIN where the master data R has been considered to be sorted with respect to access frequency, while for all other approaches R has been considered to be unsorted. The performance results are shown in Figure 8.3(a). It can be observed that for all settings of R CACHEJOIN performs slightly worse than Optimised X-HYBRIDJOIN while it performs significantly better than the other three approaches.

Performance comparisons for different memory budgets: In the second experiment the performance of all these algorithms has been tested using different memory budgets while keeping the size of R fixed (*2 million tuples*) and choosing 1 as the Zipfian exponent. Figure 8.3(b) presents the performance comparisons of all approaches. In this experiment, for all memory budgets CACHEJOIN again performs significantly better than the other three approaches, while slightly worse than Optimised X-HYBRIDJOIN.



(a) Performance comparison with 95% confidence interval while M= *50MB* and R_t varies (on log scale).

(b) Performance comparison with 95% confidence interval while R_t=*2* million tuples and M varies.



(c) Value of Zipfian exponent varies

Figure 8.3: Performance comparisons of CACHEJOIN with related join algorithms

Performance comparisons by varying skew in stream *S*: The performance of CACHEJOIN has also been tested with the other related algorithms while varying the skew in input stream S. To vary the skew, the value of the Zipfian exponent has been varied. In the experiment it is allowed to range from 0 to 1. At 0 the input stream S is uniform while at 1 the stream has a larger skew. The results presented in Figure 8.3(c) show that CACHEJOIN performs nearly equally with Optimised X-HYBRIDJOIN for uniform and moderately-skewed data. However, for largely skewed data the performance of CACHEJOIN decreases slightly, compared to Optimised X-HYBRIDJOIN. If CACHEJOIN is compared with the other three approaches, it performs better for moderately-skewed data, and this improvement becomes more pronounced for

larger-skewed data. The data for exponents larger than 1 is not presented, which would imply short tails. It is clear that the trend continues for such short tails, but the focus was on understanding the limitations of the proposed approach. For completely uniform data, however, CACHEJOIN and Optimised X-HYBRIDJOIN perform worse than MESHJOIN and R-MESHJOIN by a constant factor because the former two approaches make provisions for adaptation that remain unused while both MESHJOIN and R-MESHJOIN reduce seek time.

There are two plausible reasons behind this slightly worse performance in the case of CACHEJOIN for all the above three parameters: one is the addition of the new operation, frequency detection, which increases the processing cost for CACHEJOIN. The other reason is the unsorted master data. Although this does not affect the performance of stream-probing phase, it affects performance of the disk-probing phase because the disk-probing phase can produce a comparatively lower number of matchings than Optimised X-HYBRIDJOIN. But besides this slightly lower performance, the advantage of CACHEJOIN is that it removes the need to sort the huge amount of master data on disk, which is an important contribution.

Finally, it can be concluded that after presenting both the Optimised X-HYBRIDJOIN and CACHEJOIN algorithms one has at least the option of choosing the best alternative based on the nature of the master data. When the master data is not changing frequently, Optimised X-HYBRIDJOIN would probably be a better option than CACHEJOIN for obtaining the maximum performance. The reason for this is that the master data needs to be sorted only once before starting the execution of the algorithm. On the other hand, when the master data is changing frequently, CACHEJOIN is a more appropriate option than Optimised X-HYBRIDJOIN.

8.5.1 Cost Validation

In the second dimension of the experiments the cost for the CACHEJOIN algorithm has been validated by comparing the calculated cost with the measured cost. Figure 8.4 presents the comparisons of both costs at different memory settings. The figure demonstrates that at each memory setting the calculated cost is closely matched with the measured cost, which is evidence that the implementation of the CACHEJOIN algorithm is

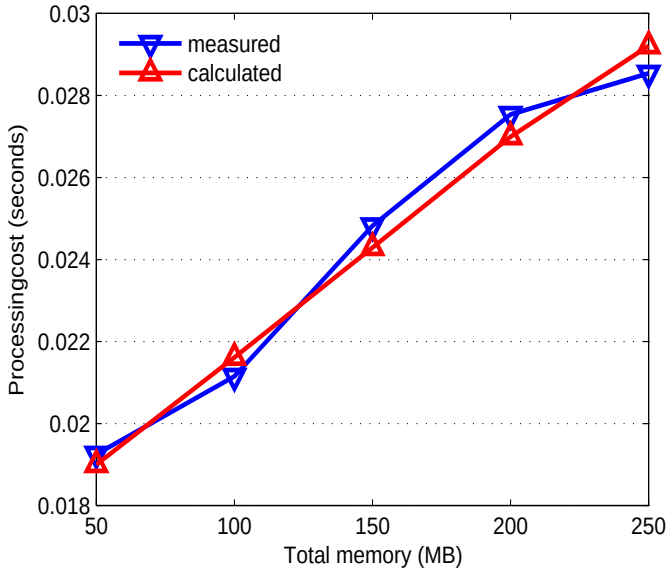


Figure 8.4: Cost validation

accurate.

8.6 Summary

In this chapter a new generalised stream-based join called CACHEJOIN has been presented that can be used to join a stream with a disk-based, slowly-changing master data table. This is called a generalised approach because of two reasons: first, it involves no assumption about sorting the master data, secondly it performs considerably better than the other approaches discussed except Optimised X-HYBRIDJOIN for all kinds of skewed, non-uniformly distributed data, as found in real-world applications. A cost model for CACHEJOIN has been presented that has also been used to precisely tune the relative size of the components of the algorithm. The provided experimental data has showed an improvement of CACHEJOIN over the other three approaches. The cost model of CACHEJOIN has been validated by comparing it with empirical results.

9

Conclusions and Future Directions

9.1 Summary of the Thesis

The primary focus of this research was to design an efficient join algorithm for the processing of stream data with persistent data. The key challenge that these kinds of algorithms normally face is to minimize the bottleneck that occurs due to disk access cost. One well known algorithm called MESHJOIN was designed previously to address this issue. Although it is a successful attempt in this direction, there are limitations, which are described and addressed in this thesis.

One issue that MESHJOIN faces is the allocation of memory to a key component called disk buffer of the algorithm. MESHJOIN did not provide data on the position of the optimal value depending on the memory size. In the literature the algorithm was tested at one particular memory settings for this component and no performance comparisons were made between this particular and default memory settings for the specified component. In Chapter 3 this research provided more data which shows that by adopting the default

settings the algorithm performs worse by only a small constant factor. Although tuning of such algorithms is good to obtain the maximum performance, at least one has the option to bypass the tuning module if a small drop in performance is acceptable.

The second issue in MESHJOIN that was explored in this research is the unnecessary dependency between the join components. Due to this dependency the algorithm is unable to optimally distribute the memory among the components. This issue was resolved in Chapter 4 after presenting a revised version of MESHJOIN called R-MESHJOIN. The R-MESHJOIN algorithm removed the complex dependency that creates the problem in MESHJOIN. A simple and accurate cost model for distributing the memory among the components was presented for R-MESHJOIN.

The other two important issues explored in this research are: (a) the approach to accessing the master data (b) dealing effectively with intermittency in stream data. Following these observations a new join algorithm called HYBRIDJOIN was introduced in Chapter 5. HYBRIDJOIN uses an efficient strategy to access the master data and can deal with intermittency in stream data effectively. A new cost model was derived for HYBRIDJOIN and the algorithm was tuned based on the cost model. A synthetic workload was also generated for testing the performance of the algorithm.

HYBRIDJOIN is an adaptive approach for joining stream data with the master data data however, it does not take into account some features of real world data. Therefore, an extended version of HYBRIDJOIN called X-HYBRIDJOIN was proposed in Chapter 6. X-HYBRIDJOIN includes all the features of HYBRIDJOIN plus some additional features. These additional features add a significant contribution in the performance of the algorithm, especially for non-uniform stream data. A cost model was derived for X-HYBRIDJOIN and it was validated empirically.

The X-HYBRIDJOIN algorithm was an effective first step in dealing with features of real world data. However, the data structures chosen for some components of X-HYBRIDJOIN were not optimal. To improve the performance of the algorithm some modifications were made and a robust Optimised X-HYBRIDJOIN algorithm was introduced in Chapter 7. The Optimised algorithm uses efficient data structures for all its components improving the performance of the algorithm significantly.

Optimised X-HYBRIDJOIN algorithm is the most optimal one among those described however, it assumes that the master data is sorted in the order of access frequency.

To remove this assumption another attempt was made in the form of CACHEJOIN in Chapter 8. CACHEJOIN is a generalised algorithm that was designed to process non-uniform stream data with unsorted disk data. It was proven through experimentation that CACHEJOIN performs better than all the other algorithms except Optimised X-HYBRIDJOIN. The cost model was derived for tuning CACHEJOIN and the cost model was validated empirically.

9.2 Achievements

The following list highlights the major achievements of this research:

1. A detailed literature review was conducted into stream-based join algorithms. This comprehensive review enabled us to find the research gap in this area.

2. A well known stream-based join algorithm MESHJOIN was explored and a number of research issues were identified. The performance of the algorithm at default memory settings for a key component was not described in the literature. A rigorous performance analysis at both optimal as well as at default settings was carried out. The experiments proved that using default settings the algorithm performs worse only by a small constant factor which is important especially when the tuning is an overhead that gains only a small increase in performance.

3. An alternative approach called R-MESHJOIN was introduced to remove the complex dependencies between the MESHJOIN components. A simple and accurate cost model was proposed for R-MESHJOIN. This enabled R-MESHJOIN to optimally distribute the memory among the components. The performance of the algorithm was also improved slightly.

4. A new algorithm HYBRIDJOIN was introduced to resolve the issues related to disk access strategy and stream intermittency. The cost of HYBRIDJOIN was calculated and the algorithm was tuned based on the cost model. A synthetic work load was designed in order to test the performance of the algorithm.

5. To make HYBRIDJOIN more specific to real world applications an extension of it was proposed in the form of X-HYBRIDJOIN. The cost model and the tuning

of the algorithm were also carried out. The experimental study proved that X-HYBRIDJOIN performs remarkably better than HYBRIDJOIN. To enhance the performance further the data structures used by the components of X-HYBRIDJOIN were improved. This was achieved in Optimised X-HYBRIDJOIN. A cost model and tuning module for Optimised X-HYBRIDJOIN were also derived.

6. Finally, a generalised approach for processing stream data with persistent data was presented in the form of CACHEJOIN. The CACHEJOIN algorithm removed one limitation of Optimised X-HYBRIDJOIN to do with sorting the master data. The cost of the algorithm was calculated and the algorithm was tuned based on the cost model. Experiments proved that CACHEJOIN performs significantly better than other approaches and only slightly worse than Optimised X-HYBRIDJOIN.

9.3 Directions for Future Research

This section describes two directions in which this research can be extended. The first is to explore further improvements to the join operators. The second is to investigate how the join operator can play a role in emerging application scenarios. In the following a number of possible examples are discussed for each direction.

9.3.1 Extensions

There are various ways that the join operators could be extended further. Two directions are presented here.

Non-equijoin

The focus of this research was on equijoins, but there are many other join operators that are widely used, such as non-equijoin. Equijoins are used in the transformation layer of ETL (Extract-Transform-Loading) where joins are executed between the foreign key attribute in stream data and the primary key attribute in master data. However, there are practical scenarios where non-equijoin operators are needed to process stream data.

Consider for example a warehouse that contains temperature-sensitive merchandise, where temperature sensors are deployed throughout the warehouse, reporting tempera-

tures at regular intervals to a main system [101]. The sensor recordings will arrive in a stream, and each product in the master data will have a temperature at which the product will expire. One likely query will raise an alarm if the temperature of any part of the warehouse is greater than the predefined temperature limit for a product stored in the warehouse. The challenge of implementing non-equijoin queries is to organise the master data appropriately. In this example, the speed of the query would be improved if a clustered index on temperature is maintained on the master data.

Categorical Attributes

Another extension is to describe a join operator that deals with categorical attributes in the master data efficiently. The current research focused on key values that were not used to describe categories. Categorical data are binary attributes e.g. gender type (male or female) or multiclass attributes e.g. product size (small, medium, large, extra large). The join operators described in this research may be suboptimal for categorical data because of I/O cost. The challenge here is to minimize this disk I/O cost by efficiently organising the master data.

9.3.2 Applications

This section presents a number of real world applications where the future of these join operators is bright.

Sensor Networks

Sensor networks are an emerging area for research in the field of computer science. A number of applications can be found in the real world that have already adopted this technology e.g. traffic monitoring systems [19], streetlight monitoring systems [60, 61], telemonitoring in health [19], weather control systems [28], environment monitoring [4, 68, 99], etc.

Consider traffic management and road monitoring systems where sensors are installed both on streets (toll monitoring) and vehicles (GPS, road conditions) [19]. These sensors generate a large amount of data which needs to be processed in near-real-time. Efficient join execution is necessary so that the required and accurate information can be sent to

all relevant destinations.

Mobile Networks

Mobile devices such as cellular phones, ipods, and laptops are becoming more and more popular everyday. According to recent statistics there are about 5 billion mobile subscribers worldwide. More than 700 mobile operators are spread across 222 countries and territories of the world [80].

Users of mobile devices communicate with each other or with network servers. They use remote data and services. Application scenarios include weather information, stock trading, electronic email, market transactions, airline information, bill payment, transport information, etc [77]. In every second a large number of requests are generated by the users and most of these requests interact with databases. The processing of this high volume of data is a challenging task requiring joins between data streams. Algorithms that are less resource intensive for query processing are sought to scale the performance.

Green Networking

The revolution of green networking is becoming a hot topic. The major objectives of green networking are [27]:

- less energy consumption,
- efficient energy utilisation,
- consideration of the environmental impact of network components from design to end of use,
- integration of networks,
- making the network more intelligent.

According to a survey [78] the power consumption of desktop computers is 39% of total power consumption. Local Area Network (LAN) is another power hungry infrastructure that includes routers, switches, modems, and cabling. According to one study [107] this power consumption can be decreased up to 90% if all desktop PCs are replaced with laptops. It can be reduced further by replacing the wired infrastructure with wireless. In

power saving environments less resource intensive algorithms are preferable for processing data.

9.4 Final Words

The key contribution of this research in the area of stream processing is to design an efficient join operator to process stream data with the master data. On the basis of various intermediate steps three robust approaches were developed in this research. The first one, called R-MESHJOIN, is efficient for uniform and continuous stream input data. The second one, called Optimised X-HYBRIDJOIN, is robust for non-uniform stream input data while the master data is sorted. The third one, called CACHEJOIN, is efficient when the stream input data is non-uniform and the master data is unsorted.

There are further extensions that can be considered, one direction is to adapt the algorithms for different kinds of data and different kinds of operators. The second direction is to consider how the algorithms can be used in different application areas.

Bibliography

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12:120–139, August 2003.

[2] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, Ninth Dover printing, tenth GPO printing edition, 1964.

[3] Chris Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006.

[4] Kurt M. Anstreicher, Marcia Fampa, Jon Lee, and Joy Williams. Maximum-entropy remote sampling. *Discrete Appl. Math.*, 108:211–226, March 2001.

[5] Francisco Araque. Real-time data warehousing with temporal requirements. In *CAiSE Workshops*, 2003.

[6] A Arasu, B Babcock, S Babu, J Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and U Srivastava. STREAM: The Stanford Data Stream Management System. *Concrete*, 2004.

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical Report 2002-57, Stanford InfoLab, 2002.

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

[9] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[10] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 336–347. VLDB Endowment, 2004.

[11] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA, 2002. ACM.

[12] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30:109–120, September 2001.

[13] Mohammad Hossein Bateni, Lukasz Golab, Mohammad Taghi Hajiaghayi, and Howard Karloff. Scheduling to minimize staleness and stretch in real-time data warehouses. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 29–38, New York, NY, USA, 2009. ACM.

[14] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28:25–40, January 1981.

[15] M W Blasgen and K P Eswaran. Storage and access in relational data bases. *IBM System*, 16(4):363, 1977.

[16] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, MDM '01, pages 3–14, London, UK, 2001. Springer-Verlag.

[17] Irina Botan, Gustavo Alonso, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Flexible and scalable storage management for data-intensive stream processing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 934–945, New York, NY, USA, 2009. ACM.

[18] Kjell Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th International Conference on Very Large Data Bases*, VLDB '84, pages 323–333, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[19] G. Brettlecker, H. Schuldt, P. Fischer, and H.J. Schek. Integration of reliable sensor data stream management into digital libraries. In *Proceedings of the 1st International Conference on Digital Libraries: Research and Development*, pages 66–76. Springer-Verlag, 2007.

[20] Robert M. Bruckner, Beate List, and Josef Schiefer. Striving towards near real-time data integration for data warehouses. In *DaWaK 2000: Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery*, pages 317–326, London, UK, 2002. Springer-Verlag.

[21] Abhirup Chakraborty and Ajit Singh. A partition-based approach to support streaming updates over persistent data in an active datawarehouse. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Abhirup Chakraborty and Ajit Singh. A disk-based, adaptive approach to memory-limited computation of windowed stream joins. In *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part I*, DEXA'10, pages 251–260, Berlin, Heidelberg, 2010. Springer-Verlag.

[23] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *Pro-*

ceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[24] Sirish Chandrasekaran and Michael Franklin. Remembrance of streams past: overload-sensitive management of archived streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 348–359. VLDB Endowment, 2004.

[25] Sirish Chandrasekaran and Michael J. Franklin. Psoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12:140–156, August 2003.

[26] Su Chen, Mario A. Nascimento, Beng Chin Ooi, and Kian-Lee Tan. Continuous online index tuning in moving object databases. *ACM Trans. Database Syst.*, 35:17:1–17:51, July 2010.

[27] Naveen Chilamkurti, Sherali Zeadally, and Frank Mentiplay. Green networking for major components of information communication technology systems. *EURASIP J. Wirel. Commun. Netw.*, 2009:35:1–35:7, January 2009.

[28] H.L. Choi and J.P. How. Efficient targeting of sensor networks for large-scale systems. *IEEE Transactions on Control Systems Technology*, (99):1–9, 2010.

[29] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient SQL-based RDF querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1216–1227. VLDB Endowment, 2005.

[30] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock: a language for extracting signatures from data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 9–17, New York, NY, USA, 2000. ACM.

[31] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 623–623, New York, NY, USA, 2002. ACM.

[32] Tamraparni Dasu, Shankar Krishnan, Suresh Venkatasubramanian, and Ke Yi. An information-theoretic approach to detecting changes in multi-dimensional data streams. In *In Proc. Symp. on the Interface of Statistics, Computing Science, and Applications*, 2006.

[33] David J. DeWitt and Jeffrey F. Naughton. Dynamic memory hybrid hash join. Technical report, University of Wisconsin, 1995.

[34] David J. DeWitt, Jeffrey F. Naughton, and Joseph Burger. Nested loops revisited. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, PDIS '93, pages 230–242, Washington, DC, USA, 1993. IEEE Computer Society.

[35] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 443–452, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[36] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, PDIS '91, pages 280–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[37] Nihal Dindar, Baris Güç, Patrick Lau, Asli Ozal, Merve Soner, and Nesime Tatbul. Dejavu: declarative pattern matching over live and archived streams of events. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 1023–1026, New York, NY, USA, 2009. ACM.

[38] Jens P Dittrich and Bernhard Seeger. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB*, pages 299–310, 2002.

[39] M. Fogiel. *Basic electricity*. Research & Education Assn, 2002.

[40] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, and Frederick Reiss. Design considerations for high fan-in systems: The hifi approach. In *Proceedings*

of Second Biennial Conference on Innovative Data Systems Research (CIDR'05), pages 290–304, 2005.

[41] MJ Franklin and T. Urhan. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[42] John E. Gaffney. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, SE-10(4):459 –464, Jul. 1984.

[43] P.B. Gibbons, B. Karp, Y. Ke, S. Nath, and Srinivasan Seshan. Irisnet: an architecture for a worldwide sensor web. *Pervasive Computing, IEEE*, 2(4):22 – 33, 2003.

[44] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Quicksand: Quick summary and analysis of network data. Technical report, 2001.

[45] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 79–88, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[46] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling updates in a real-time stream warehouse. In *ICDE 2009: Proceedings of the 25th International Conference on Data Engineering*, pages 1207 –1210, 2009.

[47] Lukasz Golab and Theodore Johnson. Consistency in a stream warehouse. In *Conference on Innovative Data Systems Research (CIDR11)*, pages 114–122, 2011.

[48] Lukasz Golab, Theodore Johnson, Nick Koudas, DivesIvesh Srivastava, and David Toman. Optimizing away joins on data streams. In *Proceedings of the 2nd International Workshop on Scalable Stream Processing System*, SSPS '08, pages 48–57, New York, NY, USA, 2008. ACM.

[49] Lukasz Golab, Theodore Johnson, J. Spencer Seidel, and Vladislav Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 847–854, New York, NY, USA, 2009. ACM.

[50] Lukasz Golab, Theodore Johnson, and Oliver Spatscheck. Prefilter: predicate push-down at streaming speeds. In *Proceedings of the 2nd International Workshop on Scalable Stream Processing System*, SSPS '08, pages 29–37, New York, NY, USA, 2008. ACM.

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

[52] Lukasz Golab and M. Tamer Özsu. Update-pattern-aware modeling and processing of continuous queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 658–669, New York, NY, USA, 2005. ACM.

[53] Hector Gonzalez, Jiawei Han, Xiaolei Li, and Diego Klabjan. Warehousing and Analyzing Massive RFID Data Sets. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 83–83, Washington, DC, USA, 2006. IEEE Computer Society.

[54] The STREAM Group. STREAM: The Stanford Stream Data Manager. Technical Report 2003-21, Stanford InfoLab, 2003.

[55] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18:3–18, 1995.

[56] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28:287–298, June 1999.

[57] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream window join: tracking moving objects in sensor-network databases. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, SSDBM '03, pages 75–84, Washington, DC, USA, 2003. IEEE Computer Society.

[58] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *SIGMOD Rec.*, 26:171–182, June 1997.

[59] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An adaptive query execution system for data integration. *SIGMOD Rec.*, 28(2):299–310, 1999.

[60] Chunguo Jing, Dongmei Shu, and Deying Gu. Design of streetlight monitoring and control system based on wireless sensor networks. In *Proceedings of 2nd IEEE Conference on Industrial Electronics and Applications, ICIEA'07*, pages 57 –62, May 2007.

[61] Chunguo Jing, Dongmei Shu, Deying Gu, and Bin Liu. Streetlight power cable monitoring system based on wireless sensor networks. volume 0, pages 1284–1288, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[62] Alexandros Karakasidis, Panos Vassiliadis, and Evaggelia Pitoura. ETL queues for active data warehousing. In *IQIS '05: Proceedings of the 2nd International Workshop on Information Quality in Information Systems*, pages 28–39, New York, NY, USA, 2005. ACM.

[63] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, VLDB '04, pages 180–191. VLDB Endowment, 2004.

[64] Won Kim. A new way to compute the product and join of relations. In *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*, SIGMOD '80, pages 179–187, New York, NY, USA, 1980. ACM.

[65] Jon Kleinberg. Bursty and hierarchical structure in streams. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 91–101, New York, NY, USA, 2002. ACM.

[66] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[67] Flip Korn, S. Muthukrishnan, and Yihua Wu. Modeling skew in data streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 181–192, New York, NY, USA, 2006. ACM.

[68] Andreas Krause, Ajit Singh, and Carlos Guestrin. Near-Optimal Sensor Placements in Gaussian Processes: Theory, Efficient Algorithms and Empirical Studies. *J. Mach. Learn. Res.*, 9:235–284, June 2008.

[69] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M.A. Shah. TelegraphCQ: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, 2003.

[70] Wilburt Labio and Hector Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 63–74, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[71] Wilburt Labio, Jun Yang, Yingwei Cui, Hector Garcia-Molina, and Jennifer Widom. Performance issues in incremental warehouse maintenance. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 461–472, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[72] Wilburt Juan Labio, Janet L. Wiener, Hector Garcia-Molina, and Vlad Gorelik. Efficient resumption of interrupted warehouse loads. *SIGMOD Rec.*, 29(2):46–57, 2000.

[73] Ramon Lawrence. Early Hash Join: a configurable algorithm for the efficient and early production of join results. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 841–852. VLDB Endowment, 2005.

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

[75] Gang Luo, J.F. Naughton, and C.J. Ellmann. A non-blocking parallel spatial join algorithm. In *ICDE'02. Proceedings of 18th International Conference on Data Engineering*, pages 697 –705, 2002.

[76] S. Madden and M.J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of 18th International Conference on Data Engineering, 2002.*, pages 555–566. IEEE, 2002.

[77] Rajeswari Malladi and Karen C. Davis. Applying multiple query optimization in mobile databases. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9*, HICSS '03, pages 294–303, Washington, DC, USA, 2003. IEEE Computer Society.

[78] S. Mingay. Green it: the new industry shock wave. *Gartner RAS Core Research Note G*, 153703:2, 2007.

[79] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 251–263, Washington, DC, USA, 2004. IEEE Computer Society.

[80] M. Moretti. Globalization of mobile and wireless communications: Bridging the digital divide. pages 19–29. Springer, 2011.

[81] M. Asif. Naeem, G. Dobbie, and G. Weber. Comparing global optimization and default settings of stream-based joins. In *VLDB Workshop (BIRTE'09)*, pages 155–170, Lyon, France, 2009.

[82] M. Asif Naeem, Gillian Dobbie, and Gerald Weber. An event-based near real-time data integration architecture. In *EDOCW '08: Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops*, pages 401–404, Washington, DC, USA, 2008. IEEE Computer Society.

[83] M. Asif Naeem, Gillian Dobbie, and Gerald Weber. HYBRIDJOIN for near-real-time data warehousing. *International Journal of Data Warehousing and Mining (IJDWM)*, 7, 2011.

[84] M. Asif Naeem, Gillian Dobbie, and Gerald Weber. X-HYBRIDJOIN for near-real-time data warehousing. In *BNCOD'11: 28th British National Conference on Databases*, Manchester, UK., 2011. Springer.

[85] M. Asif Naeem, Gillian Dobbie, and Gerald Weber. Optimised X-HYBRIDJOIN for near-real-time data warehousing. In *ADC'12: 23rd Australasian Database Conference*, pages 21–30, Melbourne, Australia, 2012. Australian Computer Society.

[86] M. Asif Naeem, Gillian Dobbie, Gerald Weber, and Shafiq Alam. R-MESHJOIN for near-real-time data warehousing. In *DOLAP'10: Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP*, pages 53–60, Toronto, Canada, 2010. ACM.

[87] M. Asif Naeem, Gillian Dobbie, Gerald Weber, and Imran Sarwar Bajwa. Efficient usage of memory resources in near-real-time data warehousing. In *IMTIC'12: Proceedings of the International Multi-topic Conference*, Pakistan, 2012. Springer.

[88] A Nguyen and A Tjoa. Zero-latency data warehousing for heterogeneous data sources and continuous data streams. In *iiWAS'2003 - The Fifth International Conference on Information Integrationand Web-based Applications Services*, pages 55–64, 2003.

[89] Vern Paxson and Sally Floyd. Wide-area traffic: the failure of Poisson modeling. *SIGCOMM Comput. Commun. Rev.*, 24:257–268, October 1994.

[90] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.E. Frantzell. Supporting streaming updates in an active data warehouse. In *ICDE 2007: Proceedings of the 23rd International Conference on Data Engineering*, pages 476–485, Istanbul, Turkey, 2007.

[91] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.E. Frantzell. Meshing streaming updates with persistent data in an active data warehouse. *IEEE Trans. on Knowledge and Data Engineering*, 20(7):976–991, 2008.

[92] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1999.

[93] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[94] Eugene J. Shekita and Michael J. Carey. A performance evaluation of pointer-based joins. *SIGMOD Rec.*, 19:300–311, May 1990.

[95] John Miles Smith and Philip Yen-Tang Chang. Optimizing the performance of a relational algebra data base interface. In *Proceedings of the 1975 ACM SIGMOD*

International Conference on Management of Data, SIGMOD '75, pages 64–64, New York, NY, USA, 1975. ACM.

[96] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34:42–47, December 2005.

[97] Mark Sullivan and Andrew Heybey. Tribeca: a system for managing large databases of network traffic. In *Proceedings of the Annual Technical Conference on USENIX*, ATEC '98, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[98] Aaron Sun, Daniel Dajun Zeng, and Hsinchun Chen. Burst detection from multiple data streams: a network-based approach. *Trans. Sys. Man Cyber Part C*, 40:258–267, May 2010.

[99] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 214–226, New York, NY, USA, 2004. ACM.

[100] Yufei Tao, Man Lung Yiu, Dimitris Papadias, Marios Hadjieleftheriou, and Nikos Mamoulis. RPJ: producing fast join results on streams through rate-based optimization. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 371–382, New York, NY, USA, 2005. ACM.

[101] Peter A. Tucker, David Maier, and Tim Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Eng. Bull.*, 26(1):33–40, 2003.

[102] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2:706–717, August 2009.

[103] Tolga Urhan and Michael J. Franklin. Xjoin: Getting fast answers from slow and bursty networks. Technical report, University of Maryland, 1999.

[104] Tolga Urhan and Michael J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *Proceedings of the 27th International Conference*

on Very Large Data Bases, VLDB '01, pages 501–510, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[105] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB '2003: Proceedings of the 29th International Conference on Very large Data Bases*, pages 285–296. VLDB Endowment, 2003.

[106] Michail Vlachos, Kun-Lung Wu, Shyh-Kwei Chen, and Philip S. Yu. Correlating burst events on streaming stock market data. *Data Min. Knowl. Discov.*, 16:109–133, February 2008.

[107] M. Webb. Smart 2020: Enabling the low carbon economy in the information age. *The Climate Group London*, 2008.

[108] A. N. Wilschut and P. M. G. Apers. Pipelining in query execution. In *Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications (PARBASE 1990), Miami Beach, FL, USA*, pages 562–562, Los Alamitos, March 1990. IEEE Computer Society Press.

[109] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS '91: Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 68–77, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[110] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[111] J. Xie and J. Yang. A survey of join processing in data streams. *Data Streams*, pages 209–236, 2007.

[112] Y. Ya-xin, Y. Xing-hua, Y. Ge, and W. Shan-shan. An indexed non-equijoin algorithm based on sliding windows over data streams. *Wuhan University Journal of Natural Sciences*, 11(1):294–298, 2006.

[113] Yin Yang and Dimitris Papadias. Just-in-time processing of continuous queries. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 1150–1159, Washington, DC, USA, 2008. IEEE Computer Society.

[114] Xin Zhang and Elke A. Rundensteiner. Integrating the maintenance and synchronization of data warehouses using a cooperative framework. *Information System*, 27(4):219–243, 2002.

[115] Xin Zhang and D. Shasha. Better burst detection. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, pages 146–146, April 2006.

[116] Shanzhong Zhu and Chinya Ravishankar. A scalable approach to approximating aggregate queries over intermittent streams. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, pages 8–94, Washington, DC, USA, 2004. IEEE Computer Society.

[117] Yunyue Zhu and Dennis Shasha. Efficient elastic burst detection in data streams. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 336–345, New York, NY, USA, 2003. ACM.

[118] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 316–327, New York, NY, USA, 1995. ACM.

A

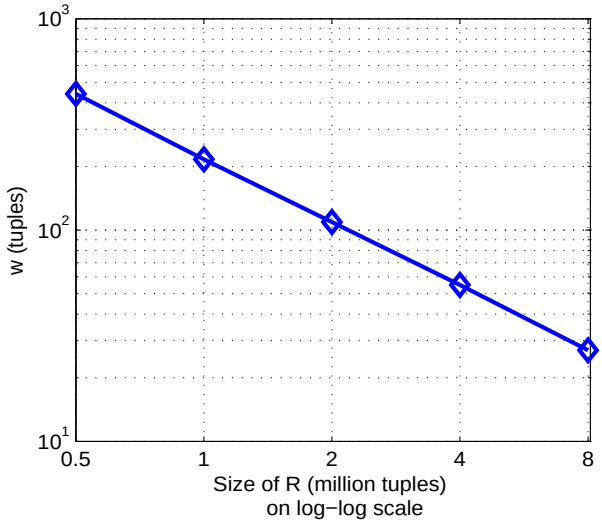
HYBRIDJOIN

A.1 Analysis of w with respect to its Related Components

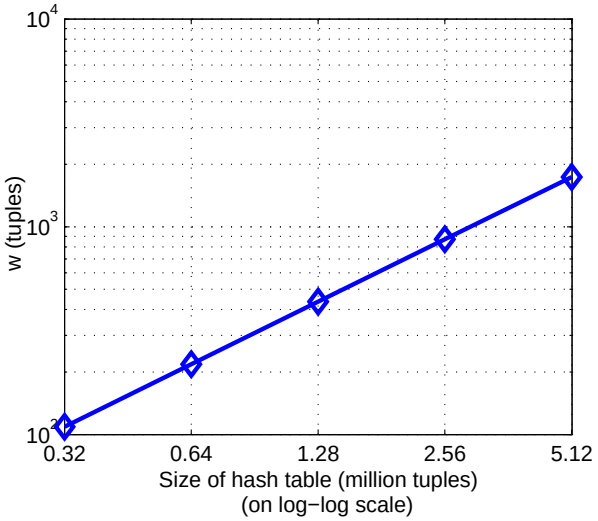
Chapter 5 provides a list of components which have a direct impact on the input size w of the algorithm. This appendix presents details of the experiments which have been conducted to observe the individual effects of each component on w.

A.1.1 Effect of the Size of the Master Data on w

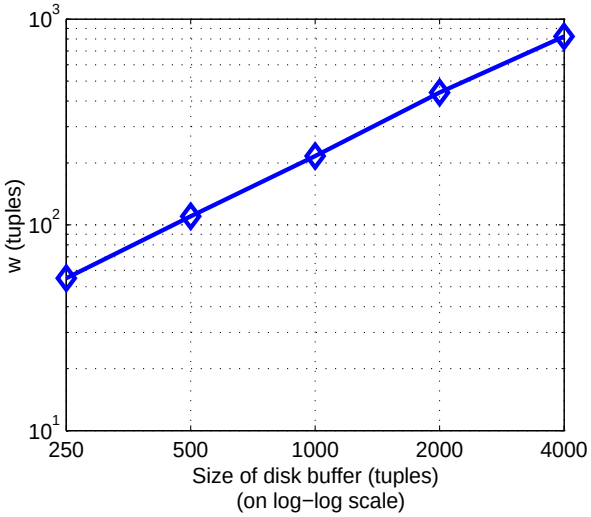
An experiment has been conducted to observe the effect of the size of the master data, denoted by R_t, on w. In this experiment the value of R_t has been increased exponentially while keeping the values for all other parameters, h_S and d, fixed. The results of this experiment are shown in Figure A.1 (a). It is clear that the increase in R_t affects w negatively. This can be explained as follows: increasing R_t decreases the probability of



(a) Effect of size of R on w



(b) Effect of size of hash table on w



(c) Effect of size of disk buffer on w

Figure A.1: Analysis of w while varying the size of necessary components

matching the stream tuples for the disk buffer. Therefore, the relationship of R_t with w is inversely proportional, represented mathematically as $w \propto \frac{1}{R_t}$.

A.1.2 Effect of the Hash Table Size on w

This experiment has been conducted to examine the effect of hash table size h_S on w. In order for us to observe the individual effect of h_S on w, the values for other parameters, R_t and d, have been assumed to be fixed. The value of h_S has been increased exponentially and w has been measured for each setting. The results of the experiment are shown in Figure A.1 (b). It can be observed that w increases at an equal rate while increasing h_S.

The reason is for this is that by increasing h_S more stream tuples can be accommodated in memory. Therefore, the matching probability for the tuples in the disk buffer with the stream tuples increases and that causes w to increase. Hence, w is directly proportional to h_S which can be described mathematically as $w \propto h_S$.

A.1.3 Effect of the Disk Buffer Size on w

Another experiment has been conducted to analyse the effect of the disk buffer size d on w. Again so that the effect of only d on w can be observed, the values for other parameters, R_t and h_S, have been considered to be fixed. The size of the disk buffer has been increased exponentially and w has been measured against each setting. Figure A.1 (c) presents the results of this experiment. It is clear that increasing d result in w increasing at the same rate. The reason for this behavior is that, when d increases, more disk tuples can be loaded into the disk buffer. This increases the probability of matching for stream tuples with the tuples in the disk buffer and eventually w increases. The relationship of w with d is directly proportional, i.e. $w \propto d$.

B

X-HYBRIDJOIN

B.1 Analysis of w based on Necessary Components

A list of the necessary components that can affect the input size w of the X-HYBRIDJOIN algorithm has been presented in Chapter 6. To visualise the individual effect of each component on w a number of experiments have been carried out. The following subsections describe these experiments one-by-one.

B.1.1 Effect of the Non-swappable Part of the Disk Buffer on w

This experiment has been conducted to analyse the effect of the size of the non-swappable part of the disk buffer, denoted by d_N, on w. Note that the non-swappable part contains the highly-used pages of R. To observe the individual effect of this component, the values of the other parameters d_S, R_t, and h_s have been fixed. The results of this experiment are shown in Figure B.1(a) where d_N has been increased sequentially and w has been measured

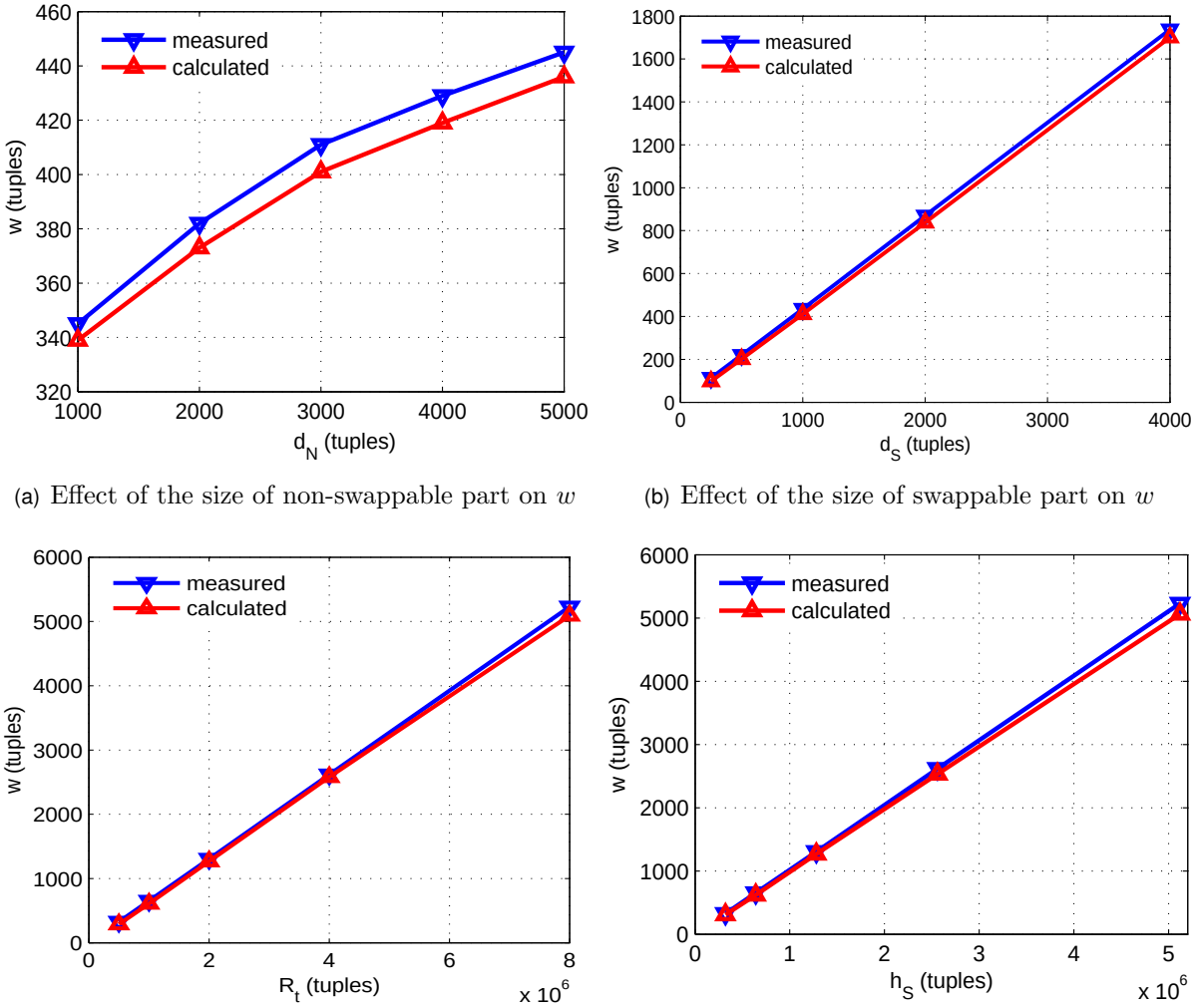
against each setting of d_N. It is clear that when d_N is increased, w also increases, but the rate of increase in w is not same as that of d_N. However, this increasing rate can be defined by some power factor. This behaviour can be explained as follows: initially increasing d_N increases the matching probability of the stream tuples with the non-swappable part of the disk buffer rapidly and this ultimately increases w. However, further increases in d_N do not increase this probability by the same rate, because the frequency of matching the stream tuples with the disk-pages, stored at later positions in the non-swappable part, decreases.

B.1.2 Effect of the Swappable Part of the Disk Buffer on w

Another experiment has been performed to analyse the effect of the size of the swappable part of the disk buffer, denoted by d_S, on w. Similar to the above the values for other parameters have been fixed. The value of d_S has been increased exponentially while measuring w. From the results shown in Figure B.1(b) it can be observed that increasing d_S, also increases w at the same rate. The reason for this behavior is that when d_S increases, more stream tuples are matched with the swappable part of the disk buffer. This increases the stream input size w. Therefore, the relationship between d_S and w is directly proportional.

B.1.3 Effect of Size of R on w

The disk-based master data is another important component that has direct influence on w. An experiment has been carried out to observe this influence empirically. In this experiment the size of R, denoted by R_t, has been increased exponentially while the remaining parameters have been assumed to be fixed. The outcomes of the experiment are shown in Figure B.1 (c). It is clear that the increase in R_t has an inverse effect on w. This behaviour can be explained as follows: when R_t is increased it equally decreases the matching probability of the stream tuples for both the swappable and non-swappable parts of the disk buffer. Therefore, w decreases.



(a) Effect of the size of non-swappable part on w

(b) Effect of the size of swappable part on w

(c) Effect of size of R on w

(d) Effect of the size of hash table on w

Figure B.1: Analysis of w while varying the size of different components

B.1.4 Effect of the Hash Table Size on w

The hash table is a larger component of the X-HYBRIDJOIN algorithm which is used to store the stream tuples in memory. This component also has a direct impact on w. To examine this impact an experiment has been conducted in which the size of the hash table h_S has been increased exponentially and w has been measured at each setting. As in the above experiments the values for the other parameters have been kept constant. The results of the experiment are shown in Figure B.1 (d). It can be observed that by increasing h_S, w increases at the same rate. A plausible reason for this is that by increasing h_S more stream tuples can be accommodated in memory. Therefore, it increases the matching probability of stream tuples with both the swappable and non-swappable parts of the disk buffer.

C

Optimised X-HYBRIDJOIN

C.1 Analysis of w_S and w_N

An experimental study has been carried out to visualize the effect of each component on w_N and w_S. Another objective is to validate mathematical formulas for both w_N and w_S. In the following the effect of each component is discussed individually.

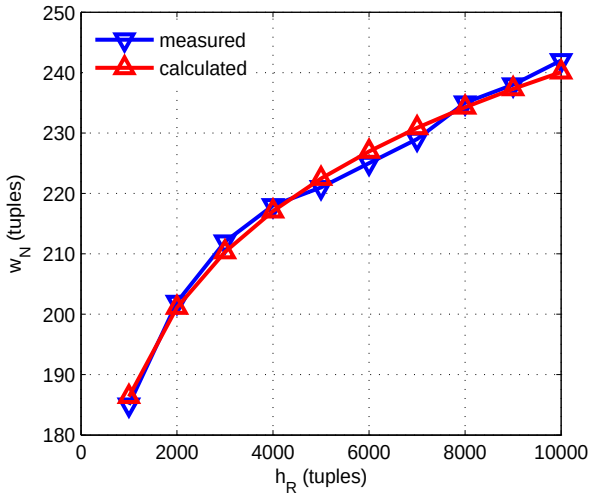
An experiment has been conducted to observe the effect of the size of the non-swappable part on w_S and w_N. In this experiment the values of other parameters (d,M, R_t) have been kept fixed. The size of the non-swappable part (denoted by h_R) has been increased sequentially and the values for both w_S and w_N have been measured against every setting of h_R. Figures C.1(a) and C.1(b) depict the results for w_N and w_S respectively. From Figure C.1(a) it can be observed that incrementing the size of the non-swappable part increases w_N. Further analysis shows that w_N increases, not at a constant rate but with some power factor. The reason for this is that initially increasing the size of the non-swappable part increases the probability of matching stream tuples rapidly. This

eventually increases w_N, as the non-swappable part contains frequently-accessed disk tuples of R. However, incrementing the size of the non-swappable part further does not make a large difference in w_N due to the decrease in the probability of matching the disk tuples (stored at, later positions in H_R) with stream tuples. In the case of w_S however, the effect of changing the size of the non-swappable part is the opposite, i.e. increasing the size of the non-swappable part decreases w_S. A plausible reason for this behaviour is that the more we increase the size of the non-swappable part, the more it covers the more frequent part of R and as a result the average matching probability p_S for the swappable part decreases.

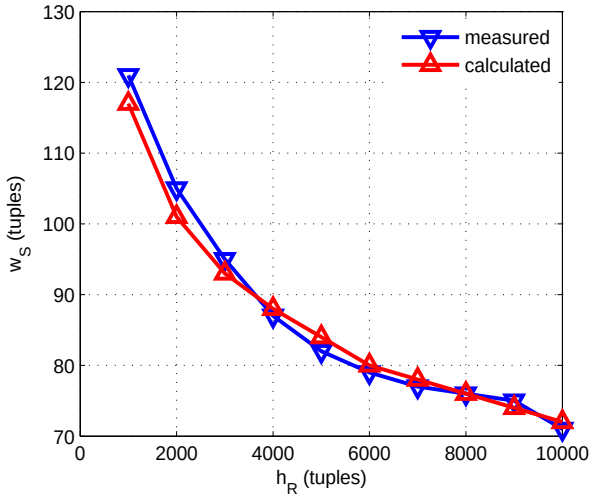
Another experiment has been conducted to observe the effect of the size of R on both w_S and w_N. In the experiment the size of R has been increased exponentially while keeping the sizes of all join components fixed. The results of this experiment are shown in Figures C.1(c) and C.1(d). As shown in the figures, a change in the size of R affects both w_S and w_N inversely, but at different rates. If the size of R is increased, w_N decreases but at a slower rate. While in the case of w_S it is approximately inversely proportional. An acceptable reason for this is that increasing the size of R decreases the probability of matching the stream tuples for both the swappable and the non-swappable parts. But in the case of the non-swappable part, even if the size of R is increased, the probability of matching for the non-swappable part is not reduced significantly due to Zipf's law.

In order to visualize the effect of the size of the swappable part on w_S, one experiment has been performed in which the size of the swappable part has been increased exponentially. The results of the experiment are shown in Figure C.1(e). It can be observed that increasing the size of the swappable part also increases w_S and at an equal rate. The reason behind this behaviour is very simple. Loading a bigger segment of R into memory increases the probability of matching accordingly. However, the change in the size of the swappable part does not affect w_N because of the independent execution of the non-swappable part.

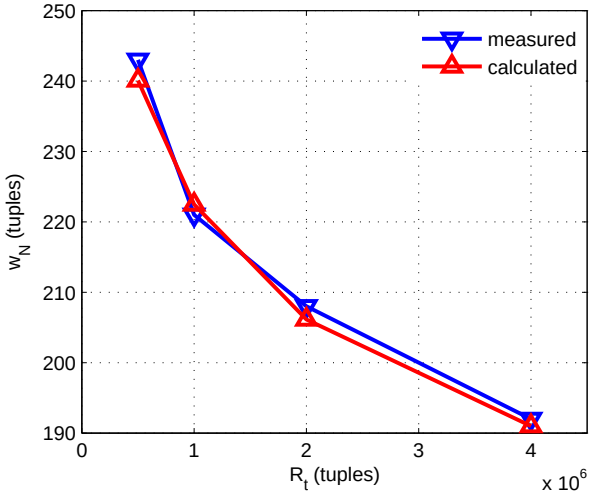
Finally, the effect of changes in the size of the hash table (used to store stream input) on w_S has also been analysed. Figure C.1(f) describes the results of the experiment where w_S increases at the same rate as that the size of the hash table. The reason is that increasing the size of the hash table allows more stream tuples to be accommodated in memory, increasing the probability of matching for the swappable part.



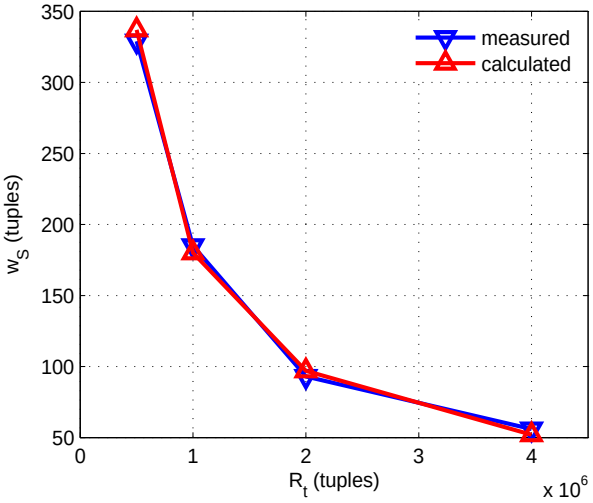
(a) Effect of size of non-swappable part on w_N



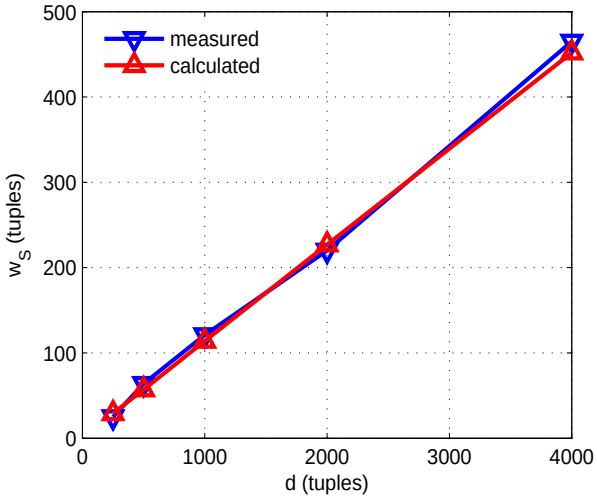
(b) Effect of size of non-swappable part on w_S



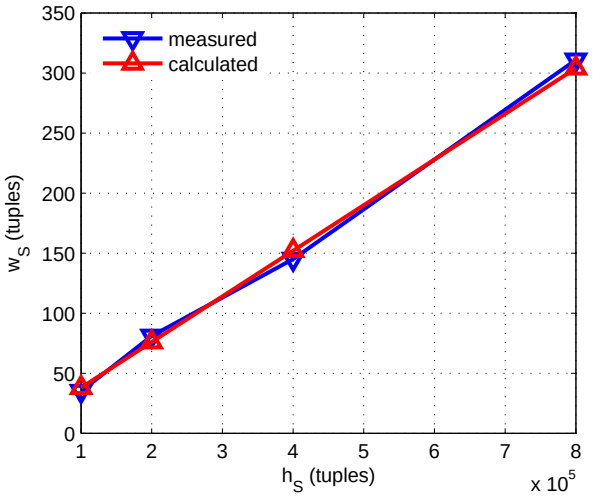
(c) Effect of size of R on w_N



(d) Effect of size of R on w_S



(e) Effect of size of swappable part on w_S



(f) Effect of size of stream-base hash table on w_S

Figure C.1: Analysis of w_S and w_N while varying the size of different components

In addition to the above analysis another objective of these experiments is to validate calculated formulas for w_S and w_N by comparing them with our empirical results. As shown in Figure C.1 the calculated results are very close to the measured results.

D

Co-authorship Forms