PROGRESSIVE QUERY PROCESSING

TOK WEE HYONG

(B.Sc.(Hons. 1), NUS)
(M.Sc., NUS)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2008

Acknowledgments

I would like to express my heartfelt gratitude to both my advisors, Stéphane Bressan and Mong-Li Lee. As advisors, both of them have patiently guided me over the years, and represents an amazing source of wisdom and inspiration. The decision to pursue a Ph.D. was seeded by Stéphane during my Honours year. He gave me the chance to visit Cornell University as a visiting graduate student. That trip left a deep imprint in my life in many ways. It provided me with an opportunity to learn and work on the open-source database management system, *Predator*, and an early version of sensors database, *Cougar*. Most importantly, it seeded the interest in pursuing a Ph.D. Throughout graduate school, Stéphane patiently guided me on the formulation of research problems, and taught me how to systematically solve them. Mong-Li taught me the art of writing research papers, and provided me with lots of opportunities through the journey. Her willingness for discussions, and insightful views on various research issues benefited me greatly.

I would also like to thank Kian-Lee Tan. Kian-Lee provided me with the opportunity during the early days of graduate school to travel to Fudan University for an exchange with the Fudan database group. That trip cemented many good friendships with Lin-Hao Xu, Ying Yan and Rong Zhang. This led to many productive discussions. The Ph.D. journey was accompanied by graduate students from the database group. In particular, Shentat Goh, Ying-Guang Li, Wei-Siong Ng and Shili Xiang enriched my life in many ways.

The job as a Teaching Assistant(TA)/Instructor provided the much needed financial support during the Ph.D. journey. The job would not have been possible without the kind support from the department. A special thank you to Aaron Tan, Eng Wee

Chionh, Gary Tan, Martin Henz, Siau Cheng Khoo, Tiow Seng Tan, and Wei Ngan Chin for giving me the chance to be a TA.

This thesis is specially dedicated to Juliet, Nathaniel and my family members. Their unconditional love gave me the strength to complete the journey. Thank you for everything!

Contents

List of Tables

List of Figures

Summary

Many join processing techniques for data streams have been proposed, the techniques are designed for a specific data model (e.g. relational), and cannot be easily generalized to other data models. In evolving data platforms (e.g. data streams, P2P, very large databases, sensor databases), the data can either be relational, spatial, high-dimensional or XML. An important criteria to support interactivity, and ensure a good user experience is the progressive production of results (if any) whenever data arrives.

In this thesis, we focus on join processing over data streams with limited memory. We focus on solving three problems on progressive, progressive and approximate, and progressive and approximate joins over a sliding window. In the first problem, we focus on progressive join processing over various data models. The problem is motivated by the observation that existing progressive join processing techniques are mostly designed for relational data streams. Thus, new progressive join processing techniques often have to be proposed for new data models. Thus, we study the problem of designing a generic framework for progressive join processing, called the Result Rate based Progressive Join (*RRPJ*) framework. The RRPJ framework offers several advantages. Firstly, it allows the generalization of the framework to handle other data models that are non-relational data (e.g. high-dimensional, spatial, XML). Secondly, as it does not require a local uniformity assumption in each of the data partitions. Thirdly, using extensive empirical evaluations, we show that RRPJ provides good performance compared with other state-of-art progressive join algorithms for the various data models. The key idea in *RRPJ* is to compute statistics based on the output of the join algorithms, and to use the statistics to determine the data that should

be kept in the limited memory in order to maximize result production. In contrast, existing works relies on statistics over the input data. Based on the RRPJ framework, we examine various instantiations of the *RRPJ* framework for four data models: such as relational, spatial, high-dimensional and XML data.

In the second problem, we focus on progressive, approximate join processing. This is motivated by the observation that due to the infinite nature of data streams, users do not need the complete results. An approximate result is often sufficient. Users expect the approximate results to be either the largest possible or the most representative (or both) given the resources available. In this problem, we studied the tradeoffs between maximizing the result quantity and quality and propose four new progressive approximate join algorithms: *ARRPJ*, *ProbHash*, *RAJ* and *RAJHash* are proposed. The former two, like *Prob*, favor quantity, the latter two favor quality. *ProbHash* improves on *Prob* on every aspect. *RAJ* and *RAJHash* produce results of significantly better quality.

In the third problem, we focus on progressive, approximate join processing over sliding window. While sliding window joins have been extensively studied, none of these used a sampling-based approach. In this thesis, we proposed a sampling-based approach for sliding window joins over data streams. In order to design progressive, approximate sliding window join algorithms, we first studied various sliding-window sampling techniques. We present both empirical and theoretical analysis for each of the sliding-window sampling techniques. Next, we propose a generic progressive, approximate sliding window join framework, which uses the sampling techniques. Through extensive performance evaluations, we show that sliding-window aware sampling-based techniques are able to produce high-quality results.

Chapter 1

Introduction

1.1 Introduction

The emergence of ubiquitous network connectivity allows data to be delivered as data streams from remote sites to be processed by applications. These new applications (e.g. sensors databases, P2P, cloud computing, XML aggregators) need to be able to process data from different data models. For example, the data that needs to be processed can range from relational, spatial, high-dimensional and XML data. In addition, the size of main memory is often limited relative to the data that needs to be processed. Indeed, this presents challenging issues in the design of a query processing algorithm framework that can be used for various data models, using limited memory. In addition, the query processing algorithms must adapt to the unpredictable nature of the query environment, and deliver results progressively.

In order to support a high-level of interactivity during query processing, the study of *progressive query processing techniques* is important. Progressive query processing techniques deliver initial results quickly, and are able to progressively produce results whenever new data arrives. Amongst the various types of queries that can be formulated, join queries is one of the most important class of queries. In this thesis, we focus on join queries. For example, in data exploration and analysis of data streams, the results needs to be presented incrementally to the users. An example of a system supporting data exploration and analysis is the *CONTROL* system [HAC$^+$99], which supports data analysis of massive data sets. In the *CONTROL* system, users are presented with initial results quickly. From the results presented, users can iteratively pose new queries or refine existing queries. This allows users to make decisions based on the initial results produced, rather than having to wait a long time for the complete results to be available.

In this thesis, we focus on the design of progressive join algorithms for data stream applications. Specifically, we study three problems. These includes progressive, progressive and approximate, and progressive and approximate joins over a sliding window. In order to solve the first problem, we propose a generic progressive join processing framework, called Result Rate-Based Progressive Join framework (RRPJ), that can deliver results incrementally using limited memory. To demonstrate the generic nature of the proposed framework for other data models, we proposed four instantiations of the framework for relational, spatial, high-dimensional and XML join processing. The focus of the work was on maximizing the quantity of results produced. In order to solve the second problem, we propose several progressive, approximate join algorithms. The focus was on maximizing either the quantity or the quality of the results produced. In order to solve the third problem, we propose several progressive, approximate sliding window join algorithms. We show how various sliding-window sampling algorithms can be incorporated within a progressive, approximate sliding join framework. We show that the results produced by sliding-window version of sampling techniques produces good quality results.

1.2 Background

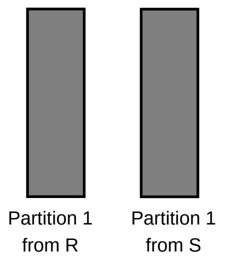
Many join processing algorithms [UFA98, UF99, DSTW02, DGR03, SW04, MLA04, XYC05, TYP$^+$05, Law05, LCKB06] have been proposed. Most of these algorithms focused on the equi-join. In order to ensure that join processing is non-blocking (or progressive), many of these equijoin algorithms leverages on the seminal work on symmetric hash join's (SHJ) [WA91]. SHJ assumes the use of in-memory hash tables, and make use of an insert-probe paradigm, which allows results to be delivered progressively to users. In an insert-probe paradigm, a newly-arrived tuple is first used to probe the hash partition for the corresponding data stream. If there are matching tuples (based on the join predicate), the matching tuples are output as results. The newly-arrived tuple is then inserted into its own hash partition. This allows results to be output immediately whenever new tuples arrives.

In order to address the issue of limited memory, many subsequently proposed progressive relational join algorithms (e.g [UFA98, UF99, MLA04, TYP^{+}05]) considered an extension of the SHJ model, where both in-memory and disk-based hash partitions are used. The extended version of the SHJ model consists of three phases: (1) Active (2) Blocked (3) Cleanup. In the active phase, data is continuously arriving from the data streams. Whenever a newly tuple arrives, it is first used to probe the hash partitions for the corresponding data stream, before it is inserted into its own hash partitions. Whenever memory is full, some of the in-memory tuples are flushed to disk to make space for new-arriving tuples. Whenever all the data stream blocks, the extended SHJ transitions into a blocked phase. During the blocked phase, data from the disk partitions are retrieved to join with either in-memory or disk-resident tuples from the corresponding data streams. This allows the delays from the blocked data streams to be hidden from the end user. In the cleanup phase, all tuples that have not been previously joined are then joined to ensure that the results produced are completed.

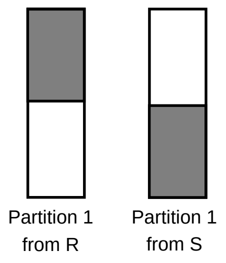
In order to maximize result throughput, a key focus of existing progressive join algorithms is to determine the set of tuples that are flushed to disk whenever memory is full. Many flushing techniques have been proposed for progressive join algorithms. These techniques can be classified as heuristic-based or statistics-based. In heuristic-based techniques, a set of heuristics govern the selection of tuples or partitions to be flushed to disk. These heuristics ranges from flushing the largest (e.g. XJoin [UF99]) to a concurrent flushing of partitions (e.g. Hash-Merge Join (HMJ) [MLA04]). In statistics-based techniques, a statistical model is maintained on the input distribution. Whenever a flushing decision needs to be made, the statistical model can be used to determine the tuples or partitions that are least likely to contribute to a future result. These tuples or partitions are then flushed to disk. Amongst the various statistical based techniques, the Rate-based Progressive Join (RPJ) and Locality-Aware (LA) model (discussed in Section 2.1) are the the state-of-art in statistic-based progressive join algorithms. RPJ rely on the availability of an analytical model deriving the output probabilities from statistics on the input data. This is possible in the case of relational equijoins but embeds some uniformity assumptions that might not hold for skewed datasets. For example, if the data within each partition is non-uniform, the RPJ local uniformity assumption is invalid.

Consider the two partitions, belonging to dataset R and S respectively, presented in Figure 1.1. The grayed area represent the data and white an empty space. The vertical axis for the rectangles represent the data values. Suppose in both Figure (a) and (b), N tuples have arrived. In Figure 1.1(a), the N tuples is uniformly distributed across the entire partitions of each dataset. Whereas in Figure 1.1(b), the N tuples is distributed within a specific numeric range (i.e. areas marked grey). Assume the same number of tuples have arrived for both cases, then $P(1|R)$ and $P(1|S)$ would be the same. However, it is important to note that if partition 1 is selected to be the partition to be kept in memory, the partitions in Figure 1.1(a) would produce results as predicted by RPJ. Whereas the partitions in Figure 1.1(b) would fail to produce any results. Though RPJ attempts to amortize the effect of historical arrivals of each relation, it assumes that the data distribution remains stable throughout the lifetime of the join, which makes is less useful when the data distribution are changing (which is common in long-running data streams).

The LA model is designed for approximate sliding window join on relational data. It relies on determining the inter-arrival distance between tuples of similar values in order to compute the utility of the tuple. Consequently, the utility of the tuple is used to guide the tuples to be flushed to disk. In the case of relational data, a similar tuple could be one that has the same value with a previous tuple. However, for non-relational data, such as spatial or high-dimensional data, the notion of similarity becomes blurred. Another limitation of the LA model is that it is unable to deal with changes in the underlying data distribution. This is because with a frequently changing data distribution, which is common in long running data streams, the reference locality, which is a central concept in the LA model cannot be easily computed. Hence, both RPJ and LA model cannot be easily extended to deal with non-relational data models.



(a) Uniform Data within partition



(b) Non-Uniform Data within partition

Figure 1.1: Data in a Partition

1.3 Motivation

As many of the existing progressive join techniques are designed for relational data model, they are not easily generalizable for other data models. As a result, new progressive join techniques, with different flushing policies need to be proposed for each type of data that needs to be processed. In addition, when processing large datasets or data streams, the amount of memory available for keeping the data is often limited. Whenever memory is full, a flushing policy is used to determine the data that are either flushed to disk partitions, or discarded. Data are flushed to disk partitions if the user is interested in the complete production of results. On the other hand, if the user is interested in approximate results, some of the in-memory data can be discarded.

This research is driven by the need to design a generic, progressive join framework that meets three objectives. Firstly, the framework must be easily generalized to different data models (e.g. relational, spatial, high-dimensional XML). Secondly, the progressive join framework must work with limited memory. Thirdly, it is important to identify the metrics that are suitable for evaluating the performance of the progressive joins. The thesis is divided into three parts.

The first part of the thesis is motivated by the need for a generic progressive join framework for which can be used in different data models. To better understand the key building blocks for a generic progressive join framework, we conducted an initial study (presented in Appendix A) for progressive spatial join processing. In the study, we observe that the building of indexes over spatial data streams is expensive. A nested-loop styled progressive join algorithms would have suffice. However, this inhibits the progressiveness of the join. Results can only be produced when the buckets used to hold the data is full. In addition, the algorithms presented in Appendix A do not consider the case of limited memory.

In order to address all these issues, we focus on SHJ-based algorithms as one of the building blocks for designing a progressive join framework. This is because the probe-insert paradigm used in SHJ-based algorithms provide the basis for producing results (if any) whenever data is available. As SHJ-based algorithms rely on hashing for probing and insertion, the challenge is to identify the appropriate hash-based data structure for each of the data models. In order to deal with limited memory, the flushing policy is one of the key ingredients for maximizing the result throughout or the quality of the approximate result subset produced. Most importantly, the flushing policy must be independent of the data model. While heuristics-based flushing policies meet the criteria of data model independence, they perform poorly compared to the statistics-based techniques. Most importantly, statistics-based techniques provide strong theoretical guarantees on the expected result output. However, existing statistic-based techniques suffer from the data model dependence. While many good statistic-based techniques have been proposed for the relational data model, none of these can be easily extended for other data models. In order to have a generic flushing policy, we observed that the goal of progressive join algorithms is on result throughput maximization. Motivated by this, we conjectured that the statistics used to determine the data that are flushed from memory should be result-driven.

The second part of the thesis is motivated by the observation that users might not need the production of complete results. Also, in data stream applications, the notion of complete results is impractical, since the data streams can be potentially infinite. When approximate results are produced, it is important to distinguish between the quantity and quality of the results. Noting that sampling-based techniques has been previously disqualified by the authors of [DGR03] without further investigations, we

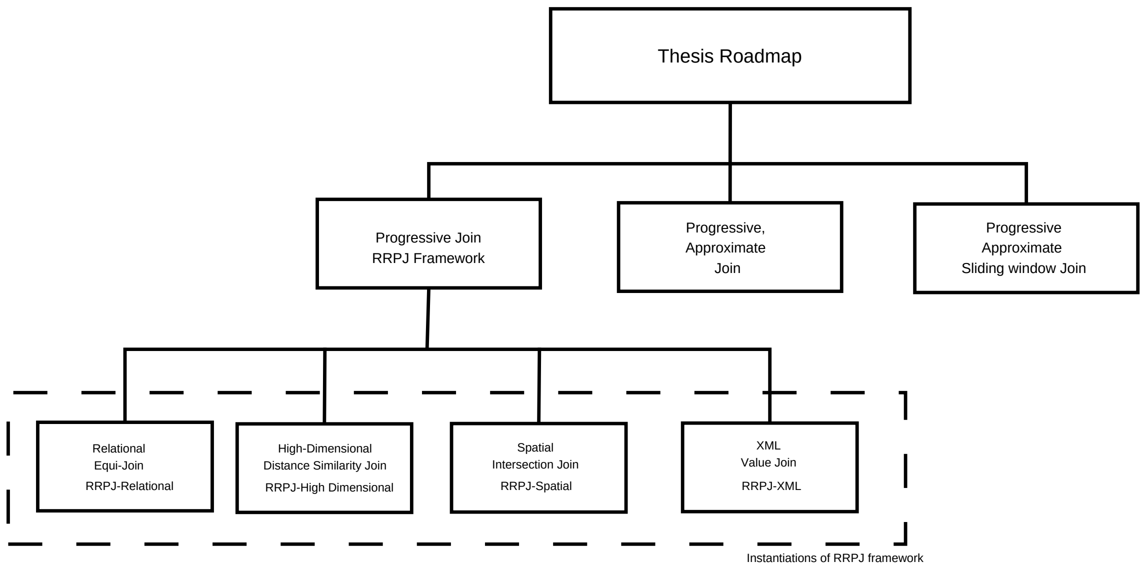

Thesis Roadmap
Progressive Join RRPJ Framework
Progressive, Approximate Join
Progressive Approximate Sliding window Join
Relational Equi-Join RRPJ-Relational
High-Dimensional Distance Similarity Join RRPJ-High Dimensional
Spatial Intersection Join RRPJ-Spatial
XML Value Join RRPJ-XML
Instantiations of RRPJ framework

Figure 1.2: Roadmap of thesis

show that this disqualification is mistaken. In the thesis, we show that a stratified sampling approach is both effective and efficient for progressive, approximate join processing.

Motivated by the success of sampling for progressive, approximate join processing, the third part of the thesis focus on using sampling-based techniques for progressive, sliding-window join processing. As sampling forms the basis for these class of algorithms, we conducted a comprehensive study on various sliding-window based sampling techniques. Using these sliding-window based sampling techniques, we propose sampling-based progressive, sliding-window join algorithms and evaluated the quality of the results produced.

1.4 Thesis Contributions and Roadmap

In this section,we discuss the contributions of the thesis, and present the roadmap on the organization of the thesis. The roadmap for the thesis is presented in Figure 1.2.

The first contribution of the thesis is a novel result-rate based progressive join framework, called RRPJ framework. The strength of the RPPJ framework it that it can be easily extended for various data models This is in contrast to non-generic approaches which are mainly for a specific data model (e.g relational). In order to demonstrate the generic nature of the proposed progressive join framework,we systematically studied various instantiations of the generic progressive join framework and evaluated their performance for the following data models: (a) Relational (b) Spatial (c) High-Dimensional and (d) XML.

In the various instantiations of the framework,we show that RRPJ is effective and efficient and is able to ensure a high result throughput using limited memory. We proposed an early version of the generic progressive join framework for spatial data, called *JGradient*. *JGradient* builds a statistical model based on the result output. The results of this research have been published in [TBL06]. Using the insights from [TBL06], we proposed a generic progressive framework, called Result-rate based progressive join (RRPJ) for relational data streams. RRPJ improves on *JGradient* in several aspects. Firstly, RRPJ take into consideration the size of each of the hash partitions. Secondly, an amortized version of RRPJ was introduced to handle changes in the result distribution from long-running data streams. The results of this research have been published in [TBL07c]. In order to show that the RRPJ can be instantiated for other data models, we studied the issues that arise from using the framework for high-dimensional data streams. We show that the high-dimensional instantiation, called *RRPJ High Dimensional* is able to maximize the results produced using limited memory.The results of this research have been published in [TBL07b]. We also showed how the RRPJ framework can be used for progressive XML value join processing. We proposed to decompose For-Where-Return (FWR) XQuery queries into a query plan that composes of twig queries and hash joins. In addition, we also proposed a result-oriented method for routing tuples in a multi-way join, called Result-Oriented Routing (*RoR*). *RoR* is used for routing tuples for join processing over multiple XML streams. The method is generic and can also be used for other data models. The results of this research have been published in [TBL08b].

To demonstrate the real-world applications of the RRPJ framework, we developed a system demo for continuous and progressive processing of RSS feeds, called *Danaïdes*. In *Danaïdes*, users pose queries in a SQL dialect. *Danaïdes* supports structured queries, spatial query and similarity queries. The *Danaïdes* service continuously processes the subscribed queries on the referenced RSS feeds and, in turn, published the query results as RSS feeds. Whenever memory is full, *Danaïdes* uses the RRPJ framework to determine the RSS feeds that are flushed to disk. The results of *Danaïdes* is a RSS feed, which can be read by standard RSS readers. The results of this research have been published in [TBL07a].

In data stream applications, users often do not require a complete answer to their query but rather only a sample. They expect the sample to be either the largest possible or the most representative (or both) given the resources available. In the second contribution, we clearly differentiated the notions of quantity and quality of results that are produced from progressive, approximate joins. Four new progressive approximate join algorithms: *ARRPJ*, *ProbHash*, *RAJ* and *RAJHash*. The former two, like *Prob*, favor quantity, the latter two favor quality. *ProbHash* improves on *Prob* on every aspects. *RAJ* and *RAJHash* produce results of significantly better quality. We conducted an extensive performance evaluation of the various progressive approximate join algorithms, and show the tradeoffs between maximizing quantity and quality. The results of this research have been published in [TBL08a].

In the third contribution,we propose a generic framework for designing sampling-based progressive sliding window joins. In order to evaluate the effectiveness of various sampling techniques we considered the use of four sliding-window based sampling techniques. These includes: *Expire* [BDM02], and 2 new sliding window sampling algorithms: *FIFO* and *WinRes*. As a baseline, we also included the conventional reservoir sampling. In order to study the effectiveness of each of these sampling techniques, we studied the performance of each of the techniques prior to incorporating them within the sliding window join framework. We present both empirical and theoretical analysis for each of the proposed sampling techniques. Next, we incorporated each of these sampling techniques in the sliding window join framework, and conduct an extensive performance evaluation. We are currently preparing a technical report based on the results of this research.

1.5 Thesis Organization

The remainder of the thesis is organized as follows: In Chapter 2,we provide a comprehensive discussion of related work. In Chapter 3, We present a generic progressive join framework. Next, we present various instantiations in which the framework can be applied. These include using the framework for relational (Chapter 4), high-dimensional (Chapter 6), spatial (Chapter 5), and XML data (Chapter 7). In Chapter 8, we propose a sampling-based approach for progressive, approximate joins. In Chapter 9, we propose a sampling-based approach for progressive sliding-window join. In Chapter 10, we conclude and present future work.

The appendices are organized as follows: In Appendix A, we present an initial study on progressive spatial joins. This summarizes the work done prior to the design of the generic join framework, and provides insights into the design of a progressive join framework for other data models. In Appendix B, we provide the XML used in the XML value join scenario for Chapter 7. As part of the thesis, we also proposed a query processing engine, called *Danaides* for aggregating RSS feeds. We present the system in Appendix C. We present the performance analysis of various sliding window sampling techniques in Appendix D.

Chapter 2

Related Work

In this chapter, we present the related work for progressive joins. Section 2.1 to 2.4 discuss the related work for progressive query processing techniques for the various data models - relational,, spatial, high-dimensional and XML. Next, we discuss the related work for data stream synopsis. Four types of data stream synopsis construction techniques are presented. These include sampling, sketches, wavelets, and histogram. We justify why sampling techniques is an attractive building block for progressive, approximate joins. In Section 2.6, we present the related work for progressive, approximate joins.

2.1 Relational Joins

Many methods [UF99, MLA04, TYP$^+$05, LCKB06] have been proposed to deal with the progressive equi-join problem on relational data streams. A recent trend amongst these methods is to make use of probabilistic models on the data distribution to determine the best data to be kept in memory.

RPJ [TYP$^+$05] is a multi-stage hash-based join algorithm that was proposed for joining data that are transmitted from remote data sources over a unreliable network. Similar to hash-based join algorithms like XJoin, RPJ stores the data into partitions. Each partition consists of two portions, one residing in memory and the other on disk. Whenever a new data arrives, RPJ computes the hash value based on the join attribute, and uses this to probe the corresponding partition to identify matching tuples. The RPJ algorithm consists of several stages. The stages are as follows: (1) Memory-to-Memory (2) Reactive. In the Memory-to-Memory stage (mm-stage), arriving data are joined with the in-memory tuples from the other data set. Whenever the memory overflows, selected tuples are flushed to the disk partitions. The Reactive Stage is triggered whenever the data source blocks. It consists of two sub-tasks: (i) Memory-Disk (Md-task) and (2) Disk-Disk (Dd-task). In the Md-task, data that are in memory are joined with their corresponding partitions on disk. And in the Dd-task, data that are on disk are joined with the corresponding partitions from the other data sets on disk. One of the key idea in RPJ is to maximize the number of results tuples by keeping tuples that have higher chances of producing results with the tuples from the corresponding data set in memory. An *Optimal Flush* technique was proposed to flush tuples that are not useful to disk. This is achieved by building a model on the tuples' arrival pattern and data distribution. Whenever memory becomes full, the model can be used to determine probabilistically which tuples are least likely to produce tuples with the other incoming data, and hence flushed from memory to disk. RPJ computes $p_i^{arr}(v)$, which denotes the probability that the next incoming tuple would be from data source i, and has the value v. Using the arrival probabilities, the RPJ strategy is illustrated by the following example. The tuples from two remote data sources R and S, are continuously retrieved, and joined. The join condition is $R.a = S.a$, the domain of the join attribute, a, is {2,4,6,8}. The arrival probabilities for R are: $p_R^{arr}(2) = 10\%$, $p_R^{arr}(4) = 15\%$, $p_R^{arr}(6) = 4\%$ and $p_R^{arr}(8) = 6\%$; whereas the arrival probabilities for S are: $p_S^{arr}(2) = 5\%$, $p_S^{arr}(4) = 20\%$, $p_S^{arr}(6) = 30\%$ and $p_S^{arr}(6) = 10\%$. At the instance when memory overflows, each of the data sources has 2 tuples for each value in memory. Suppose n_{flush}=6 tuples need to be flushed from memory. Since the arrival probability for $p_R^{arr}(6) = 4\%$ is the smallest, we will need to flush 2 *S-tuples* with the value 6 from memory (i.e. these S-tuples would be least likely to produce results since the corresponding R-tuples do not arrive as often compared to other tuples). Since n_{flush}=6, we would need to flush 4 more tuples from memory. We consider the next smallest arrival probability. In this case, $p_S^{arr}(2) = 5\%$ is the smallest. Thus, we flush 2 *R-tuples* with the value 2 from memory. Finally, we consider $p_R^{arr}(8) = 6\%$, and flush 2 *S-tuples* with the value 8 from memory.

[LCKB06] observes that a data stream exhibits reference locality when tuples with specific attribute values has a higher probability of re-appearing in a future time interval. Leveraging this observation, a Locality-Aware (LA) model was proposed, where the reference locality caused by both long-term popularity and short-term correlations are captured. This is described by the following model: $x_n = x_{n-i}$ (with probability a_i); $x_n = y$ (with probability b, where 1 ≤ i ≤ h and b + $\sum_{i=1}^{h} a_i = 1$. y denotes a random variable that is independent and identically distributed (IID) with respect to the probability distribution of the popularity, P. Using this model, the probability that a tuple t will appear at the n-th position of the stream is given by $Prob(x_n = t|x_{n-1}, ..., x_{n-h}) = bP(t) + \sum_{j=1}^{h} a_j \delta(x_{n-j}, t)$ ($\delta(x_k, t) = 1$ if $x_k =$ t, and it is 0 otherwise). Using the LA model, the marginal utility of a tuple is then derived, and is then used as the basis for determining the tuples to be flushed to disk whenever memory is full.

2.2 Spatial Joins

In this section, we discuss various types of spatial join processing techniques that have been proposed. In addition, we have also conducted an extensive survey on continuous query processing on spatial data, which is presented in [Ibr06].

Spatial index structures such as R-tree [Gut84], R+-tree [SRF87], R*-tree [BKSS90] and PMR quad-tree [NS87] were commonly used together with spatial joins. In [BKS93], Brinkhoff et al. proposed a spatial join algorithm which uses a depth-first synchronized traversal of two R-trees. The implicit assumption is that the R-trees has already been pre-constructed for the two spatial relations to be joined. A subsequent improvement to the synchronized traversal was proposed by [HJR97], called *Breadth First R-tree Join* (BFRJ). By traversing the R-tree level by level, BFRJ was able to perform global optimization on which are the next level nodes to be accessed, and hence minimize page faults. In [LR94], a seeded tree method for spatial joins was proposed. It assumes that there is a pre-computed R-tree index for one of the spatial relations. The initial levels of the R-tree index is then used to provide the initial levels (i.e. *seeds*) for the dynamically constructed R-tree of the corresponding spatial relation. An integrated approach for handling multi-way spatial join was proposed in [MP99]. Noting that the seeded tree approach performs poorly when the fanout of the tree is large to fit into a small buffer, [MP99] also proposed the *Slot Index Spatial Join* to tackle the problem.

The use of hashing was explored in [LR96, PD96]. In [LR96], the Spatial Hash Join (SHJ) was proposed to compute the spatial join for spatial data sets which has no indexes pre-constructed. Similar to its relational counter-part, the spatial hash join consists of two phases: (1) Partitioning Phase and (2) Join Phase. In the Partitioning Phase, a spatial partitioning function first divides the data into outer and inner buckets. In order to address issues due to the *coherent assignment* problem, a multiple assignment of data into several buckets was adopted. This allows two bucket pairs to be matched exactly once, and reduces the need to scan other buckets. In the join phase, the inner and outer buckets are then joined to produced results. The Partition Based Spatial-Merge (PBSM) method proposed in [PD96] first divides the space using a grid with fixed-sizes cells (i.e. tiles). These tiles is then mapped to a set of partitions. The data objects in the partitions are then joined using a computational geometry based plane-sweeping technique. Noting that in a plane-sweeping approach, only the data objects that are along the sweeping line are needed in memory, the Scalable Sweeping-Based Spatial Join (SSSJ) [APR$^+$98] was proposed.

Spatial join algorithms based on other novel data structures have also been proposed. The *Filter Trees* [SK96], a multi-granularity hierarchical structure, was used as an alternative to R-trees and its variants. Noting that techniques such as PBSM and SHJ requires replication of data, the Size Separation Spatial Join (S^3J) [KS97] was proposed by building incomplete *Filter Trees* on-the-fly and using them in join processing.

Existing spatial join processing techniques focus on reducing the number of I/Os for datasets that reside locally. None of these proposed techniques are optimized for delivering the initial results quickly, and do not consider the case where spatial data are continuously delivered from remote data sources.

2.3 High-Dimensional Distance-Similarity Joins

Many efficient distance similarity joins [SSA97, KS00, BBBK00, BBKK01, KP07] have been proposed for high- dimensional data. To facilitate efficient join processing, similarity join algorithms often relies on spatial indices. R-trees (and variants) [Gut84], X-tree [BKK96] or the ϵ-kdb tree [SSA97] are commonly used. The Multi-dimensional Spatial Join (MSJ) [KS00, KS98] sorts the data based on their Hilbert values, and uses a multi-way merge to obtain the result. The Epsilon Grid Order (EGO) [BBKK01] orders the data objects based on the position of the grid-cells. Another related area is the K-nearest Neighbor (KNN) [BK03, BK04]. The focus is not the efficiency of processing of local high-dimensional datasets. The Multi-page Index (MUX) method [BK04], uses R-trees to reduce the CPU and I/O costs of performing the KNN join. GORDER [XLOH04] uses Principal Component Analysis (PCA) to identify key dimensions, and uses a grid for ordering the data.

The main limitation of conventional distance similarity join algorithms is that they are designed mainly for datasets that reside locally. Hence, they are not able to deliver results progressively.

2.4 XML Query Processing

XML (Extensible Markup Language) is now a standard for data dissemination and interchange. In most application domains, XML data feeds or data streams is commonly being used. In this section, we discuss various types of spatial join processing techniques that have been proposed. In addition, we have conducted an extensive survey on progressive and continuous query processing on XML data, which is presented in [Par08].

To seize the opportunity created by the availability of such a wealth of network accessible timely data, modern application need the capability to effectively and efficiently process queries to XML data streams. In XML, concrete XML query languages, such as XPath and XQuery, express both structural and predicate constraints on the XML document/stream.

One good representation of the structural constraints is twig queries. A twig query is a tree-pattern query that specifies the structural relationships (parent/child or ancestor/ descendant) between the nodes. Existing XML query processing techniques has focused on the efficient processing of twig queries. Our focus in the thesis is the progressive processing of XML joins, expressed using join predicates.

We classify existing XML query processing techniques by considering the following factors: (1) Non-streaming vs Streaming and (2) Handle single vs multiple XML documents/streams.

2.4.1 Non-streaming and Single XML document

Non-streaming techniques [BKS02, LCL04, CLT+06] processes disk-resident XML data. These techniques focused on the efficient processing of twig queries. The assumption made by these techniques is that a labeling scheme is available. The labeling scheme encodes the structural relationships within the XML documents. Common labeling schemes that have been used include Region [BKS02] and Dewey-based [TVB+02, LLCC05] encoding. The non-streaming algorithms rely on these encodings to efficiently answer the queries. In order to compute the results, the algorithms need to wait for all the intermediate results to be produced before the results of the twig queries can be computed. Due to the need for prior labeling of the XML data and the need to wait for all the intermediate results to be produced before results are available, these techniques are not suitable for processing XML data streams.

Non-streaming techniques [FHK+03, PWLJ04, RSF06] for processing XQuery have also been proposed. In [FHK+03], a transducer-based XML Query Processor translates XQuery to an intermediate form, known as XML Stream Machine (XSM). XSM is then translated into C code which is compiled and executed. [PWLJ04] transforms XQuery into a Tree-Logical Class (TLC) algebra expression, which is then used as the basis for evaluating the XQuery query.

2.4.2 Streaming and Single XML document

Streaming techniques for processing XPath and XQuery queries include [LMP02, FHK$^+$03, PC03, OKB03, LA05, CDZ06]. In [FHK$^+$03], the BEA/XQRL processor was proposed to support pipelined execution by using an iterator model over the data stream. [LA05] proposed transformation techniques to enable XQuery queries to be evaluated in one-pass. In addition, [LA05] proposed code generation techniques (from the XQuery queries) to handle user-defined aggregates and recursive functions. [CDZ06] proposed the *TwigM* machine, an efficient non-blocking method for evaluating twig queries over XML data streams. *TwigM* assumes an input sequence of SAX events (i.e. startElement, endElement), and uses a stack-based structure to compactly encode the solutions to the twig join. The output consists of XML fragments. None of these techniques considered XML query processing over multiple XML data streams. In this thesis, we make use of multiple *TwigM* machine for twig matching.

2.4.3 Streaming and Multiple XML documents/streams

[HDG$^+$07] proposed a Massively Multi-Query Join Processing (MMQJP) technique for processing value joins over multiple XML data streams. Similar to our approach, MMQJP consists of two phases: XPath Evaluation and Join Processing phase. In the XPath evaluation phase, the XML data streams are matched and auxiliary information stored as relations in a commercial database management systems (DBMS) - Microsoft SQL Server. The auxiliary information are then used during the join processing phase for computing results. Thus, MMQJP can only deliver results when the entire XML documents have arrived. In addition, MMQJP have no control over the flushing policy due to its dependence on the commercial DBMS. In contrast to MMQJP, our proposed technique delivers results progressively as portions of the streamed XML documents arrived.

In addition, a physical algebra for XQuery was proposed in [SFMS07]. The algebra allows XML streaming operators to be intermixed with conventional XML and relational operators in a query plan. This allows pipelined plans to be easily defined. [SFMS07] do not consider memory management issues.

2.5 Data Stream Synopsis

Data stream applications need to process large amount of data over an extended period of time. As the computational resources and memory available for processing the data is much smaller relative to the size of the data streams, one-pass algorithms are often desired. Users often do not require complete answers to their queries, and are satisfied with approximate answers. The approximate answers can either be a subset of the complete answer, or an estimation of one or several measured quantities. It is also important to provide guarantees of the quality of the approximate answers.

In order to support approximate query processing, synopsis are often used for summarizing the entire data stream and used to provide approximate answers to the queries. Various approximate query processing techniques which rely on synopsis have been proposed for various types of queries: aggregation queries (e.g. *quantile* [GK01], *heavy hitters* [MM02] and *distinct counts* [Gib01]) and join queries [DGR03, DGR05, AKLW07].

[Agg07] provides a comprehensive survey on synopsis construction in data streams and identified five desirable properties for building an effective synopsis. Firstly, the synopsis must be generalizable for various applications. Secondly, the algorithms used for synopsis construction and maintenance need to be one-pass algorithms. Due to the large amount of data that needs to be processed, each tuple in the data stream can be accessed once. Thirdly, the synopsis must be compact. The size of the synopsis must be relatively smaller compared to the size of the data stream. Fourthly, the synopsis must be robust and provide guarantees on the quality of the approximation. Finally, the synopsis must be able to adapt to the varying data distribution of the data streams.

In this section, we survey various synopsis construction techniques. These include sampling (subsection 2.5.1), sketches (subsection 2.5.2), wavelets (subsection 2.5.3) and histograms (subsection 2.5.4). In each of the subsections, we discuss the strengths and limitations of each of the synopsis construction techniques.

2.5.1 Sampling

Simple random sampling [Coc77] is a method for selecting n out of a population of N data items, such that it is equi-probable to select any of the $\begin{pmatrix} N \\ n \end{pmatrix}$ distinct samples. Sampling algorithms [EN82, Knu81, Vit84] have been proposed where the value of N is known. In data stream applications, as the data can be continuously arriving over an extended period, the value of N cannot be pre-determined. In order to solve the sampling problem of maintaining a sample from an unknown N over data streams, several sampling techniques have been proposed. These includes reservoir sampling [Vit85], concise sampling [GM98], chain sampling [BDM02] and min-wise sampling [NGSA04].

Reservoir Sampling

Reservoir sampling maintains an unbiased sample of n tuples in a data stream. Assume that t tuples have arrived. When $t \leq n$, then the tuple is added to the reservoir (i.e. sample). When $t > n$, the reservoir sampling technique needs to determine the tuple to be replaced. This is achieved by randomly generating a value, v, between 1 to t. If $v > n$, then the *t-th* tuple is discarded. Else, the *t-th* tuple is used to replace the *v-th* tuple in the reservoir. It is shown in [MB83, Agg07] that the reservoir sampling technique maintains an unbiased simple random sample at any point in time.

Concise Sampling

Concise sampling [GM98] was proposed to increase the number of distinct values that can be stored in a sample. Consequently, this helps to improve the quality of the sample maintained. In a concise sample, a uniform random sample of value/count pairs are maintained. For each distinct value v which appear m times (m > 1) in the data stream, it is represented as are maintained as a value/count pair {v,m}. If $m = 1$, then only a singleton with value v is maintained. It is shown in [GM98] that the quality of a concise sample is either equivalent or exponentially better than other existing sampling techniques.

Moving Window Sampling

[BDM02] further noted that in many applications, recent data are more interesting than expired data. Data expires when they are no longer valid in a window (e.g time-based or count-based windows). To address the problem of data expiration in moving windows over streaming data, the chain sampling algorithm was proposed. Chain sampling, an extension of reservoir sampling, maintains a sample size of n tuples by having n independent samples of size 1. While it was shown in [BDM02] that it is an effective technique for dealing with expiration, chain sampling suffers from several problems. Firstly, [BDM02] did not show how duplication of tuples can be prevented in the n independent sample of size 1. Secondly, chain sampling maintains a chain of indexes of replacement tuples. Thus, the check to determine whether a newly arrived tuple is the replacement tuple can be expensive. Thirdly, we need to determine the inclusion probability into each of the n samples independently. If n is large, the cost of computing the inclusion probability can be large.

Min-wise Sampling

Min-wise sampling [NGSA04] was proposed for sampling a sensor network uniformly at random. In min-wise sampling, each tuple is assigned a random tag, with a value between 0 and 1. The key idea in min-wise sampling is that since the tag value is generated uniformly, each item is equi-probable of being assigned the tag with the smallest value. Assume that t tuples have arrived and the sample size to be maintained is n ($t > n$). The n tuples with the smallest tag values are selected to be included in the sample.

Discussion

Sampling is an attractive method for constructing synopsis. Firstly, samples can be easily constructed and maintained. Secondly, sampling provides an unbiased estimate of the entire data stream. Existing sampling techniques are one-pass algorithms which can be proved theoretically that they maintain simple random samples at any point in time. Thirdly, as the samples contain the actual tuples from the data stream, they can be easily used to answer a broader set of queries. In contrast, other synopsis construction techniques transform the tuples into a summarized form, which limits the type of queries that they can be used in. Finally, sampling techniques are independent of the data model. This allows samples of various data models (e.g. XML, spatial, high-dimensional data, etc) to be constructed easily.

One of the limitations of sampling is that it cannot be used to provide approximate answers for aggregation queries. For example, an aggregation query might require the count of the number of distinct tuples in a data stream. However, since a sample contains an approximation of the entire data stream at any point in time, it is difficult to determine whether a newly arrived tuple is unique w.r.t to the sample.

2.5.2 Sketches

A sketch is a randomized projection of data into a new space. Using the projected representation, the sketch provides a compact summary of the data stream, and can be used to compute several useful properties of the data stream. As sketches can be incrementally maintained, they are commonly used in data stream applications to provide approximate answers. The applications of sketches include: counting the number of distinct elements, estimation of Euclidean distance between the values from two data streams, point, range and inner product queries.

FM Sketch

The notion of sketches was first introduced in [FM83, FM85] as probabilistic counting algorithms for database applications. The probabilistic counting algorithms are used to estimate the number of distinct elements in a large dataset. We refer to this family of sketches as Flajolet-Martin sketches (FM Sketches). In FM sketches, a uniform hash function, *h(t)* is first used to map a tuple t to a value in the range 0 to 2^L-1 (inclusive). Given that $y = \sum_{k \geq 0} bit(y,k) 2^k$ (y geq 0) , *bit(y,k)* denotes the *k-th* bit y. *p(y)* is used to denote the position of the least significant 1-bit in the binary representation of y, as follows:

$$p(y) = \begin{cases} \min\limits_{k \geq 0}(bit(y,k)) & y > 0 \\ L & \text{y} = 0 \end{cases} \tag{2.1}$$

In [FM85], it is further observed that if *h(x)* is uniformly distributed, the pattern $0^k1\ldots$ appears with probability $\frac{1}{2^{(k+1)}}$.

Using these observations, a FM sketch, *FM*, is represented as a bit vector of length *L*. *FM* is initialized to all Os. When a new tuple t arrives, we set the bit corresponding to *h(t)* to be 1. The number of distinct values in the data stream, *d*, can be estimated as $c2^R$, where c is a constant value, and R denotes the position of the rightmost zero in *FM*. In order to improve the accuracy of the FM sketches, multiple hash functions can be used.

AMS Sketch

AMS sketches [AMS96, AGMS99] are synopsis which uses randomized technique to estimate the size of the self-join, *SJ(A)*, for a relation R with respect to a join attribute A. AMS sketches offer strong probabilistic guarantees using only logarithmic space $|dom(A)|$.

[AMS96] provide a generalization of counting, and introduced the notion of frequency moment. Frequency moments provide useful statistics for estimating different properties of the data. Given that S = (s_1, s_2, …, s_i) denotes a sequence of tuples, where $s_i \in Z$. Let V be the set of values that are observed in S, and m_v to denote the number of occurrences of the value v in S (i.e. $v \in V$). The frequency moment, F_k, is defined as the sum of k-powers of m_b, as follows: $F_k = \sum\limits_{i=1}^{V} m_i^k$. Several interesting can be derived for various k values. F_0 is the number of distinct elements in S and F_1 is the size of S.

The key idea in AMS Sketches is to make use of an unbiased estimator, denoted as Y, as an approximation to the value of *SJ(A)*. In addition, as Var(Y) is sufficiently small, it ensures a good estimation for the value of *SJ(A)*. In order to build the AMS sketch, a family of 4-wise independent {-1, +1} random variables, denoted by ξ_i.

In order to compute Y, Y is defined as X^2, where $X = \sum\limits_{i \in dom(A)} f(i)\xi_i$, where *f(i)* is the frequency vector of $R.A$. In order to compute X, the value of X is initialized to 0. The value of ξ_i is added to X whenever the *i-th* value of A is observed in the data stream.

Count-Min Sketch

Count-Min sketches [CM05] are synopsis that uses a combination of counting (i.e.) and finding minimum (min) operations to provide high-quality estimation to a broad set of queries. A Count-Min (CM) sketch with parameters (ϵ, δ) is represented as a two-dimensional array of width w and depth d. Each element of the array maintains a counter, which is initialized to zero. Using parameters (ϵ, δ), $w = \left\lceil \frac{e}{\varepsilon} \right\rceil$ and $d = \left\lceil \frac{1}{\delta} \right\rceil$. In addition, d hash functions, $h_1 \ldots h_d$, are chosen from a pairwise-independent family. When a new value v arrives, each of the counters corresponding to (j, $h_j(v)$) is incremented by 1. Using a series of *count* and *min* operations, [CM05] showed how the CM sketches can be used for providing estimation of point, range and inner product queries.

Multi-dimensional Sketches

In [DGR04], sketches for spatial data are proposed. The AMS-based sketches are used to provide high quality estimation to spatial join and range queries. The notion of dyadic atomic sketches for a two-dimensional dataset is introduced. Given a spatial object represented as a minimum bounding rectangle (MBR),the key idea in [DGR04] is to maintain sketches for the whole rectangles, horizontal and vertical edges and the corner points of the rectangles. Using these sketches, [DGR04] further showed how they can be used for providing estimation to the spatial join between intervals and rectangles. In addition, the technique was generalized for providing estimation for the join of hyper-rectangles.

Discussion

Sketches offer several advantages. Firstly, sketches are able to provide a good estimation of the results to various queries using limited space. Secondly, the size of the space is sublinear with respect to the data size. Thirdly, sketches are easy to maintain and often require linear updating time.

In sketches, the original data is not stored. Instead, only the representation of the data in the transformed sketch space is maintained. As the original data is not available, one of the limitations of sketches is that they can only be used for aggregation queries. While sketches can be used to estimate the join result size, they cannot be used for approximating the results of join queries. In addition, each type of sketch is usually designed for pre-specified aggregation computation. For example, FM sketches are used for the distinct count problem, and AMS sketches are used for self-join estimation. As noted by [CM05], during data stream processing, multiple aggregates are often required. Hence, if each kind of sketch can be used for a specific aggregation computation, multiple sketches will need to be constructed. The need to maintain multiple sketches is expensive.

2.5.3 Wavelets

Wavelets [Gra95] are synopsis which provides multi-resolution representations of the data. Wavelets have been used extensively as a data decomposition tool in various applications. In [MVW98], wavelet histograms were used for selectivity estimations. [CGRS01] uses wavelet for approximate query processing. In data mining applications, the use of wavelets have also been extensively studied [LLZO02]. Wavelets have also been used for aggregate computations for static datasets [VW99]. In [pCF99], wavelets are used to reduce the dimensionality of the time series datasets. The first few wavelet coefficients are then indexed using an R-tree index. The index is used to support range and nearest neighbour queries computation.

[GKMS03] proposed the use of L_2-minimal wavelet synopses for aggregate computation over data streams for a single measure. [GKS04] extended the work to include support for aggregation over multiple measures. In [PBF03], the *AWSOM* (Arbitrary Window Stream mOdeling Method) method is proposed to automatically discover interesting patterns and trends in sensors databases.*AWSOM* uses wavelets to represent the sensors data, and make use of linear regression models to capture the correlations between the wavelet coefficients. [GH05] showed the one-pass construction of wavelet synopsis of data from data streams for non-euclidean measures

Amongst the various wavelet construction techniques that have been proposed, the Haar wavelet [BGG97] is commonly used in data stream processing. This is because Haar wavelet uses a simple wavelet basis, which allows it to be easily implemented. Multiple resolution views of the data can be computed using a combination of averaging and differencing computations. Haar wavelets provide a good approximation of the data. The construction of Haar wavelet requires linear time w.r.t to the size of the dataset. Most importantly, the preservation of Euclidean distance between the original data and the transformed wavelet representation allows similarity computation to be done in the transfromed space.

The Haar wavelet transform computes the average and differences using the values of discrete function in order to obtain the summarized value in various resolution. Given a discrete function *f(x)* = (16 18 24 22). Figure 2.1 shows the Haar transform at various resolutions. In order obtain the value 17 in resolution 2, the average of 16 and 18 is computed. Similarly,, to obtain the value 23 in resolution 2, the average of 24 and 22 is computed. In addition, the coefficients are computed by computing the differences between (16 18) and (24 22) respectively, and dividing it by 2 to obtain (-1 1).

Resolution	Averages	Coefficients
4	(16 18 24 22)	
2	(17 23)	(-1 1)
1	(20)	(-3)

Table 2.1: Example of Haar Transform

Haar transform can also be computed using matrix multiplication between the transpose of the discrete function and the Haar transformation matrix H,

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.2}$$

The Haar transformation is given in Equation 2.3.

$$\begin{bmatrix} x_0^1 \\ d_0^1 \\ x_1^1 \\ d_1^1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{2.3}$$

At resolution 2, the Haar transformation matrix is given as

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \tag{2.4}$$

Using the discrete function *f(x)* = (16 18 24 22) given earlier, we can compute the next resolution (i.e. Resolution 2) Haar transformation as follows:

$$\frac{1}{2} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} 16 \\ 18 \\ 24 \\ 22 \end{bmatrix} = \begin{bmatrix} 17 \\ -1 \\ 23 \\ 1 \end{bmatrix} \tag{2.5}$$

Using the *H*, the Haar transformation at resolution 1 is computed as follows:

$$\frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 17 \\ 23 \end{bmatrix} = \begin{bmatrix} 20 \\ -3 \end{bmatrix} \tag{2.6}$$

This produces the value of the last average (i.e. 20) and the coefficient -3.

Discussion

One of the advantages of using wavelets is that the original data can be approximately reconstructed using the wavelet synopsis. [MU05] noted that if the data contains many similar values, the wavelet transformation usually results in wavelet coefficients with very small values. When these small value coefficients are removed, it have minimal impact on the errors between the reconstructed data and the original data.

One of the disadvantages of wavelet transformation (e.g. Haar wavelets) is that they are efficient for dataset sizes that are multiple of 2. The transformation from the original data to the wavelet synopsis results provides a ‘non-smooth’, ‘ladder-like’ approximation of the original data. [GH05] also noted that wavelet approximation for the basis often uses half the difference between the left and right hand side of the basis vectors. While this method has been shown to be optimal for the Euclidean-based error measure, it is not optimal for L_m metric error measures.

2.5.4 Histograms

Histograms are synopsis which are constructed by partitioning the data distribution into several buckets. Each of the buckets provides a count (i.e. frequency) of the number of tuples within the value range for the bucket. Histograms have been to provide selectivity estimations to query optimizers, approximate query processing and time series similarity computation. Various histogram construction techniques have been proposed over the years. These include: EquiWidth [Koo80], EquiHeight [PSC84], MaxDiff, Compressed, End-Biased and V-Optimal [IP95, PIHS96], MHist [PI97] and approximate V-Optimal histograms [GSW04]. [Ioa03] provides a comprehensive discussion on the history of histogram.

Histograms that can be dynamically maintained in data stream applications have also been proposed. [GKS01, GKS06] provide a detailed discussion of the use of histograms for data streams applications. As noted by [GKS06], the accuracy of histograms is dependent on an effective partitioning of the data into buckets, which minimizes a given error measure. Amongst the many error measures, the V-Optimal measure, which computes the sum of squares of the errors at every point i, is commonly used. The V-Optimal measure is as follows: $\sum\limits_{i}(x_i - \hat{x}_i)^2$. [GK02] proposed a fixed window histogram construction algorithm for data streams, which allow near optimal histograms to be computed on-demand. The technique was used for approximate range aggregation and time series similarity query processing.

Many variants of histogram construction have also been proposed. These variants leverage on the advantages of other synopsis such as sampling, wavelets and sketch-based methods. [CMN98] uses sampling prior to histogram construction. [MVW98] proposed the use of wavelet-based histograms. In [MVW98], Haar wavelets are used to decompose the data. After decomposition, the B Haar coefficients with the largest absolute (normalized) value are used. In addition, [MVW98] showed how the wavelet histograms can be maintained over data streams. [TGIK02] proposed the use of sketches for constructing V-optimal histograms over data streams. The key idea in [TGIK02] is to consider the data distribution and histograms as point in high-dimensional space. Given any histogram, the sketch is computed. The histogram sketch is then compared against the sketch for the data distribution. An optimal histogram can be obtained by finding the histogram which has a sketch that is very similar to the sketch for the data distribution.

Discussion

One of the advantages of using histograms is that they are relatively easy to use for answering a broad set of queries (e.g. range queries, size of join result). One of the disadvantages of conventional histograms is that the data distribution for the tuples within each bucket is not stored. In addition, the data distribution within a bucket is often assumed to be uniform. Consequently, this impacts on the accuracy of the estimation that is computing using the histogram. Another disadvantage of histograms is that the accuracy of the histogram is often dependent on finding a good partitioning of the data into buckets. In addition, optimal histogram construction often requires the use of offline, dynamic programming techniques. This presents challenges in using histograms in data stream applications. Several techniques have been proposed [MVW98, GKS01, TGIK02, GKS06] to tackle these challenges.

2.5.5 Summary

In this subsection, we discuss the key ideas behind various type of synopsis that are used in data stream processing. In addition, we also present the advantages and disadvantages for each of the synopsis construction techniques.

In our thesis, we make use of sampling techniques for progressive, approximate join processing. Amongst the different type of synopsis that are discussed, sampling provides a subset of the original dataset. This allows the approximate results of the join queries to be produced incrementally as data arrives. In contrast, while sketches have been used to provide estimation of aggregation queries, as well as estimating the size of the join result (e.g. self-join), it is not able to produce an approximation of the actual join results. This is due to the randomized projection of the original data to the sketch space. While wavelets allow the re-construction of the data based on the varying resolutions that are produced, it is expensive to re-construct the data in an online manner. Indeed, this motivates the use of sampling techniques for progressive, approximate join processing.

2.6 Progressive, Approximate Joins

In the data stream literature, various approximate query processing techniques have been proposed for aggregation queries (e.g. *quantile* [GK01], *heavy hitters* [MM02] and *distinct counts* [Gib01]) and join queries [DGR03, DGR05, AKLW07]. In this thesis, we focus on progressive, approximate join queries where the results are streamed out to the user as soon as they are produced.

2.6.1 Progressive Joins

Progressive relational equi-join algorithms [TYP$^+$05, TBL07c] studied the problem of producing complete results over data streams. In order to work with limited memory these algorithms need to flush tuples to disk whenever memory is full. These disk-resident tuples are then joined at subsequent phases in order to produce the complete join results. The goal of these algorithms is to maximize results production, as well as ensure high result throughput during join processing. If we retain only the in-memory processing phase, these algorithms are suitable for approximate join processing. In this thesis, we modify one of the state-of-art progressive relational equi-join algorithm, RRPJ [TBL07c], and show how it can be used for progressive approximate join processing.

2.6.2 Approximate Joins

In conventional databases, several techniques [OR86, Olk93, CMN99] have been proposed for approximate join processing. These techniques construct a fixed size random sample of the results for a relational query. The underlying assumption is that indices are available for one or both of the datasets, or statistics on the data distribution known apriori. [OR86, Olk93] assumes that indexes are available to facilitate efficient random access to the data. Given two relations S_1 and S_2, it randomly chooses a tuple t_{S1} from S_1, and determines whether t_{S1} should be included into the sample by computing its inclusion probability. If t_{S1} is included, a tuple t_{S2}, with the same join attribute, is then randomly chosen from S_2 and joined with t_{S1}. Noting this, [CMN99] proposed a generalized technique for sampling the results of join queries which do not require indices to be pre-constructed. In addition, Surajit et al. noted that for skewed data distributions, the random sampling of results from join queries could cause a worst-case scenario in which no join results are available. None of these works deal with data streams. In this thesis, we show the impact of the worst-case scenario for data stream processing.

Worst-case Scenario

In this section, we describe a static and dynamic case for the worst-case scenario noted in [CMN99] for approximate joins. In the extreme scenario, the data distributions for the relations to be joined are skewed.

It was noted that when the data distributions are skewed, the join of the samples would not produce any results [CMN99]. Given two relations R_1(A,B) and R_2(B,C), where A, B and C are attributes of the relations. Each relation consists of N tuples. Figure 2.1 shows the data in each of the relations. Suppose we obtain two random samples S_{R_1} and S_{R_2} from R_1 and R_2 respectively. The likelihood that the value b_1 is selected and included in sample S_{R_1} will be very low. Similarly, the likelihood that the value b_2 is selected included in sample S_{R_2} is also very low. Thus, if we compute the samples first, and then compute $S_{R_1} \bowtie S_{R_2}$, the results will be empty. We refer to this as the static case. In the static case, we assume that all the data is available, and we first create the samples from each of the relations. The join is performed only when the samples are created.

Figure 2.1: Extreme Case

A	B
a_1	b_1
a_2	b_2
a_3	b_2
a_4	b_2
...	...
a_N	b_2

R_1

B	C
b_2	c_1
b_1	c_2
b_1	c_3
b_1	c_4
...	...
b_1	c_N

R_2

In contrast, we consider the dynamic case for data streams applications. In the dynamic case, we progressively build the sample and perform the join at the same time. Assume that we maintain two samples S_{R_1} and S_{R_2}, each of size n. In this example, we set $n = 2$. Reservoir sampling [Vit85] is used to maintain the two samples. When the tuple $R_1(a_1,b_1)$ arrives, we first probe S_{R_2} to find any tuples that can be joined. Since S_{R_2} is empty, no results are produced. We then insert $R_1(a_1,b_1)$ into S_{R_1}. Next, when the tuple $R_2(b_2, c_1)$ arrives, we probe S_{R_1}. Similarly, no results are produced. $R_2(b_2, c_1)$ is inserted into S_{R_2}. When the tuple $R_1(a_2,b_2)$ arrives, the probe of S_{R_2} will generate one result. It is then inserted into S_{R_1}. Similarly, when the tuple $R_2(b_1,c_2)$ arrives, it will join with the tuple in S_{R_1}. As the two samples are now full, when the next tuple arrives for R_1, it will have a probability of 2/3 to replace a randomly selected tuple in the reservoir. Since there are only two tuples in the sample, the probability that the rare tuple $R_1(a_1,b_1)$ is replaced is 1/3. When the size of the reservoir is large, the probability that the rare tuple will be replaced in the dynamic case will be small. Thus, for the dynamic case, join results will still be produced even for skewed distributions.

2.6.3 Progressive Approximate Joins

Several progressive approximate join algorithms [DGR03, DGR05, AKLW07] have been proposed for data stream applications. In [DGR03] and its extended version

[DGR05], the motivation was the maximization of the result subset produced. A reference theoretical algorithm, called *OPT-offline*, was proposed. The algorithm presents an optimal scenario in which the *MAX-subset* error measure is minimized. They cannot be used for online applications. In order to deal with the online case, two heuristics, *PROB* and *LIFE*, were proposed to maximize the expected output size. The focus of the work was on maximizing the result output size of the approximate join, and assumes the availability of a fast CPU for join processing. Given two streams, S_1 and S_2, the priority of a tuple from S_1 is computed based on the arrival probability of tuples from S_2. Priority queues are used for storing the in-memory tuples. Whenever a tuple from S_1 arrives, it will need to scan the entire priority queue of S_2 (and vice-versa). Our work differs in two aspects. Firstly, we show how auxiliary data structures (i.e. hash-based priority queues) can be used to minimize the need to scan all the tuples in memory. Secondly, we show that maximizing the output size of the result does not necessarily ensure good result quality. We quantify the notion of result quality, and propose a technique that is able to deliver good-quality results progressively. Though [AKLW07] also studied the use of reservoir sampling over memory-limited join, the focus of the work was on how to balance between the memory allocated for join buffers and the reservoir. In addition, [AKLW07] do not progressively output results.

Chapter 3

Generic Progressive Join Framework

In this chapter, we present the issues that need to be considered for designing a generic progressive join framework. These include the need for a data structure to support efficient frequent probe-insert, and a data-model independent flushing policy to determine the data to be flushed to disk whenever memory is full

3.1 Building Blocks for Generic Progressive Join Framework

3.1.1 Data Structures

We focus on data structures used for hash-based joins. It is important to note that even though progressive join algorithms based on sort-merge paradigm (e.g. [DSTW02]) exist, these algorithms are not able to deliver initial results quickly, as results can only be produced when the data structure (i.e. sweep area) used is sufficiently full before sorting can be performed.

A data structure, D, used to store data and support the join algorithm must have the following required properties: (1) Correctness and (2) Completeness. It must ensure that the results produced are correct with respect to the join predicate used.

In addition, it must ensure that the complete result set can be produced. In addition, a desirable property is for the data structure is for it to be *minimal.* This means that the data structure must ensure that the minimum number of partitions are scanned in order to find the complete result set. For example, if we make use of a data structure, D, for storing data from two data sources R and S. Given a tuple t from R, if all the partitions from S need to be scanned in order to identify the result set, then D is not minimal.

D divides the data space into equal-sized partitions, each denoted by P_i, where i denotes the *i-th* partition. Whenever a new data object o arrives, a partitioning function f determines the partition which o belongs to. Formally, $f(o) \rightarrow I$, where I denotes a set of partitions, N denotes the total number of partitions, and $\{1, ..., N\} \in I$. Ideally, $|I| = 1$ (i.e each object is assigned to a single partition).

For relational data, this can be easily achieved by choosing a good hash function, f, which assigns each data object into a single partition. For spatial data, we make use of the same spatial partitioning function used in Spatial Hash Joins [LR96]. Each spatial object is replicated into the grid-cells in which it intersects. As observed in [LR96], the replication is necessary to allow pairwise joins between partitions from each data source. For high-dimensional data (i.e. n-dimension), each object is inserted into the partition (i.e. grid-cell) in which is falls into.

Next, we introduce the notion of a correspondence function, κ. κ maps a partition P to the set of partitions from the other data stream that need to be scanned. Formally, $\kappa(P) \rightarrow J$, where J denotes the set of partition(s) that need to be scanned from the other data stream. This is illustrated in Figure 3.1, where a partition P is mapped to a partition in the other data grid. It is important to note that it is possible that a partition from one grid need not necessary map to the same partition in the other data grid.

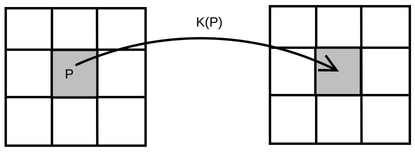


Figure 3.1: Correspondence Function, κ

Data Type	Data Structure, DS	Partitioning Function, f	$\|I\|$	κ
Relational	Hash-based partitions	Modulo	1	Identity
Spatial	2-dimensional Grid	Insert each object into the grid-cells in which it intersects	$\geq$1	Identity
High Dimensional Data	n-dimensional Grid	Insert each object into the grid-cell it falls into	1	Non-identity

Table 3.1: Various Data Structures

For both relational data and spatial data, κ is usually the identity correspondence (i.e. I = J), which is necessary to ensure that only pairwise partitions (one from each of the data streams) are scanned in order to identify the complete result set. This helps to prevent redundant scanning of partitions which will not yield any results. If the partitioning functions used for each of the data stream are not the same, then κ is non-identity. For high-dimensional data, κ is non-identity. This is because, for each grid-cell from one data stream, the grid-cell that are contains data that are epsilon-distance needs to be scanned in order to find the complete result set. Table 3.1 summarizes the data structure, and the partitioning function used for each data type.

3.1.2 Flushing Policy

Whenever memory becomes full, the flushing policy determines the tuples to be flushed to disk. The goal of the generic progressive join framework is to design flushing policies that are independent of the data model used. Consequently, this allows the generic progressive join framework to be easily instantiated for other data models easily. In contrast, flushing policies which are dependent on the input data distribution and the type of join predicates cannot be easily generalized.

3.2 Progressive Join Framework

We consider the problem of performing a join J between two datasets R and S, which are transmitted from remote data sources through an unpredictable network. Let R and S be denoted by $R = \{r_1, r_2, \ldots, r_n\}$, and $S = \{s_1, s_2, \ldots, s_m\}$, where r_i and s_j denotes the *i-th* and *j-th* data objects. The join predicate is denoted by J_{pred}. Formally, $(r_i,\ s_j)$ is reported as the result if r_i and s_j satisfies J_{pred}. The goal of is to deliver initial results quickly and ensure a high result-throughput.

The general form of a progressive join algorithm presented in Algorithm 1. In Algorithm 1, we assume that there are two remote data sources, R and S. The in-memory data structures used to store the data objects from R and S are denoted by DR and DS respectively. *endOfStream(...)* determines whether data from the stream has completely arrived. This is usually indicated by an end-of-stream marker sent by the remote data source. *isBlocked(...)* determines whether data from the stream is blocked (i.e. data did not arrive for a user-defined duration). *ProcessUnJoinedData()* determines the data that has not been previously joined and joins them to produce results. *select(R,S)* gets the data from either of the data streams to be processed.

In the In-Memory phase, whenever a new data object t arrives, it is used to probe (line 9 or line 12) the corresponding data structure to identify all the data objects in DS_s that joins with it. Once the probe is completed, t is then inserted into the data structure used to store the in-memory data (line 10 or line 13). During the insertion of t, the algorithm needs to check whether the memory is full. If it is full, data needs to be flushed to disk. This is determined by a flushing policy.

When both the data sources block (lines 2-5), the algorithm moves into the *Blocking Phase*. In order to produce results during this phase, the join algorithm joins the in-memory data with the on-disk data. When all the in-memory data has been joined, the algorithm would need to join disk-resident data from both the data sources. This allows results to be produced even though both data streams are blocked.

In the Cleanup phase (line 17), data which have not been joined in the prior phases are joined. These include joining in-memory data with disk-resident data and

Algorithm 1 Generic Progressive Join

```
while ( !endOfStream(R) and !endOfStream(S) ) do
    if ( isBlocked(R) and isBlocked(S) ) then
        //Blocking Phase
        ProcessUnJoinedData()
    end if
    //In-memory Phase
    tuple $t$ = select(R,S)
    if ( t.src == R) then
        DS.probe(t)
        DR.insert(t)
    else if (t.src == S) then
        DR.probe(t)
        DS.insert(t)
    end if
end while
//Cleanup Phase
CleanUp()
return (Results tuples from the join)
```

disk-resident data with disk-resident data. These ensure that the complete result set is produced. Due to the multiple invocation of the various phases, duplicate results would be produced. These duplicates are removed using online duplicate elimination methods which has been extensively described in [UF99] and [TYP$^+$05].

3.2.1 Result-Rated Based Flushing

In this section, we present a flushing policy which maintains statistics over the result distribution, instead of the data distribution. This is motivated by the fact that in most progressive join scenarios, we are concerned with delivering initial results quickly and maintaining a high overall throughput. Hence, the criteria used to determine the tuples that are flushed to disk whenever memory becomes full should be 'result-motivated'. We refer to join algorithms that make use of the result-rate based flushing policy as Result-Rate Based Progressive Join (RRPJ).

Whenever memory is full, we compute the Th_i values (i.e value computed by formula given in Equation 3.3) for all the partitions. Partitions with the lowest Th_i values will then be flushed to disk, and the newly arrived tuple inserted. The main difference between the *RRPJ* flushing and *RPJ* is that the Th_i values are reflective of the output (i.e. results) distribution over the data partitions. In contrast, the RPJ values are based on input the data distribution.

To compute the Th_i values (computed using Equation 3.3), we track the total number of tuples, n_i (for each partition), that contribute to a join result from the probes against the partition. Intuitively, RRPJ tracks the *join throughput* of each partition. Whenever memory becomes full, we flush n_{flush} (user-defined parameter) tuples from the partition that have the smallest Th_i values, since these partitions have produced the least result so far. If the number of tuples in the partition is less than n_{flush}, we move on to the partition with the next lowest Th_i values.

Given two timestamps t_1 and t_2 ($t_2 > t_1$)and the number of join results produced at t_1 and t_2 are n_1 and n_2 respectively. A straightforward definition of the throughput of a partition i, denoted by Th_i, is given in Equation 3.1.

$$Th_i = \frac{n_2 - n_1}{t_2 - t_1} \text{ (version 1)} \tag{3.1}$$

From Equation 3.1, we can observe that since $(t_2 - t_1)$ is the same for all partitions, it suffice to maintain counters on just the number of results produced (i.e. n_1 and n_2). A partition with a high Th_i value will be the partition which have higher potential of producing the most results. Moreover, it is important to note that Equation 3.1 does not take into consideration the size of the partitions and its impact on the number of results produced. Intuitively, a large partition will produce more results. It is important to note that this might not always be true. For example, a partition might contain few tuples, but produces a lot of results. This partition should be favored over a relatively larger partition which is also producing the same number of results. Besides considering the result distribution amongst the partitions, we must also consider the following: (1) Total number of tuples that have arrived, (2) Number of tuples in each partition, (3) Number of result tuples produced by each partition and (4) Total results produced by the system. Therefore, we use an improved definition for Th_i, given below.

Suppose there are P partitions maintained for the relation. Let N_i denote the number of tuples in partition i ($1 \leq \mathrm{i} \leq P$), and R_i denote the number of result tuples produced by partition i. Then, the Th_i value for a partition i can be computed. In Equation 3.2, we consider the ratio of the results produced to the total number of results produced so far (i.e. numerator), and also the ratio of the number of tuples in a partition to to the total number of tuples that have arrived (i.e. denominator).

$$Th_i = \left(\frac{R_i}{\sum_{j=1}^{P} R_j}\right) / \left(\frac{N_i}{\sum_{j=1}^{P} N_j}\right) = \frac{Ri \times \sum_{j=1}^{P} N_j}{\sum_{j=1}^{P} R_j \times N_i} \quad \text{(version 2)} \tag{3.2}$$

Since the total number of results produced and the total number of tuples is the same for all partitions, Equation 3.2 can be simplified. This is given in Equation 3.3.

$$Th_i = \frac{Ri}{N_i} \quad \text{(version 2 - after simplification)} \tag{3.3}$$

Equation 3.3 computes the Th_i value w.r.t to the size of the partition. For example, let us consider two cases. In case (1), suppose $N_i = 1$ (i.e. one tuple in the partition) and $R_i = 100$. In case (2), suppose $N_i = 10$ and $R_1 = 1000$. Then, the Th_i values for case (1) and (2) are the same. This prevents large partitions from unfairly dominating the smaller partitions (due to the potential large number of results produced by larger partitions) when a choice needs to be made on which partitions should be flushed to disk.

3.2.2 Amortized RRPJ (ARRPJ)

In order to allow RRPJ to be less susceptible to varying data distributions, we introduce Amortized RRPJ (ARRPJ). ARRPJ assumes that the arrival order of data is not random w.r.t to the entire stream, but to a specific time interval. Thus, the set of tuples that are received is not a random sample of the entire stream.

Suppose there are two partitions P_1 and P_2, each containing 10 tuples. If P_1 produces 5 and 45 result tuples at timestamp 1 and 2 respectively, the Th_1 value is

5. If partition P_2 produces 45 and 5 result tuples at timestamp 1 and 2 respectively, the Th_2 value for P_2 will also be 5. From the above example, we can observe that the two scenarios cannot be easily differentiated. However, we should favor partition P_1 since it is obviously producing more results than P_2 currently. This is important because we want to ensure that tuples that are kept in memory are able to produce more results because of its current state, and not due to a past state.

To achieve this, let σ be a user-tunable factor that determines the impact of historical result values. The amortized RRPJ value, denoted as A_i^t, for a partition i at time t is presented in Equation 3.4. r_i^j denotes the number of results produced by partition i at time j. When $\sigma = 1.0$, then the amortized RRPJ is exactly the same as the RRPJ. When $\sigma = 0.0$, then only the latest RRPJ values are considered. By varying the values of σ between 0.0 to 1.0 (inclusive), we can then control the effect of historical RRPJ on the overall flushing behavior of the system.

$$A_i^t = \frac{\sigma^t r_i^0 + \sigma^{t-1} r_i^1 + \sigma^{t-2} r_i^2 + ... + \sigma^1 r_i^{t-1} + \sigma^0 r_i^t}{N_i} = \frac{\sum_{j=0}^{t} \sigma^{(t-j)} r_i^j}{N_i} \tag{3.4}$$

3.3 Summary

In this chapter, we have presented the two key ingredients for the designing generic progressive join algorithms. These include using a data structure that supports a probe-insert paradigm, as well as a generic flushing policy, which builds a statistical model on the output (i.e. result) distribution. Using these key ingredients, we propose the generic Result-Rate Based Progressive Join (RPPJ) framework. In addition, we also show an amortized version of RRPJ can be used to handle changing data distributions for long running data streams.

We show the various instantiations for the RRPJ framework for different data models in the next few chapters. These instantiations include relational (Chapter 4), high-dimensional (Chapter 6), spatial (Chapter 5), and XML data (Chapter 7). In each of these instantiations, we discuss the issues that needs to be considered for the specific data model.

Chapter 4

Progressive Relational Join

In this chapter, we present the instantiation of the RRPJ framework for the processing of progressive equijoin for relational data streams. The algorithm is of the X- and symmetric hash join family. Its originality is twofold.

Firstly, the algorithm implements a replacement strategy for main memory partitions that estimates the probability of partition to produce results directly from the observation of output statistics. Previous proposals, such as the RPJ and LA algorithms, have attempted to analytically construct such a model from the statistics on the input streams. We showed that our algorithm is equivalent to RPJ in the cases for which RPJs performance was evaluated by its inventors (we use the same data sets). We showed that our algorithm significantly outperforms RPJ, when the uniformity hypothesis necessary to the estimation by the RPJ algorithm does not hold. We therefore showed that our algorithm is globally better than RPJ empirically.

Secondly, we proposed an adaptive version of our algorithm that makes use of amortization in order to incrementally weight out the influence of past statistics. The same principle can be incorporated in previously proposed algorithms such as RPJ and LA. This allows the algorithm to cater for changes over time in the input data distributions. We showed that this technique leads to significant performance increase in some cases. However, the results we obtained compel further studies in order to understand the impact of the different parameters.

We consider the problem of performing a relational equijoin between two relational

Dataset Parameter	**Default Values**
Number of Tuples Per Page	85
Available Memory	1000 pages
Domain of Join attribute	[1, 10000]
Tuple Inter-arrival	0.001s
Dataset Size (Relation R1 + Relation R2)	2 million tuples
Percentage of tuples flushed	10%

Table 4.1: Experiment Parameter

datasets, which are transmitted from remote data sources through an unpredictable network. Let the two sets of relational data objects be denoted by $R = \{r_1, r_2, \ldots, r_n\}$, and $S = \{s_1, s_2, \ldots, s_m\}$, where r_i and s_j denotes the *i-th* and *j-th* data object from the remote data source respectively. When performing a relational equijoin, with join attribute A, a result is returned when $r_i.A$ is equal to $s_j.A$. Formally, (r_i, s_j) is reported as the result if $r_i.A$ is equal to $s_j.A$. The goal is to deliver initial results quickly and ensure a high result-throughput.

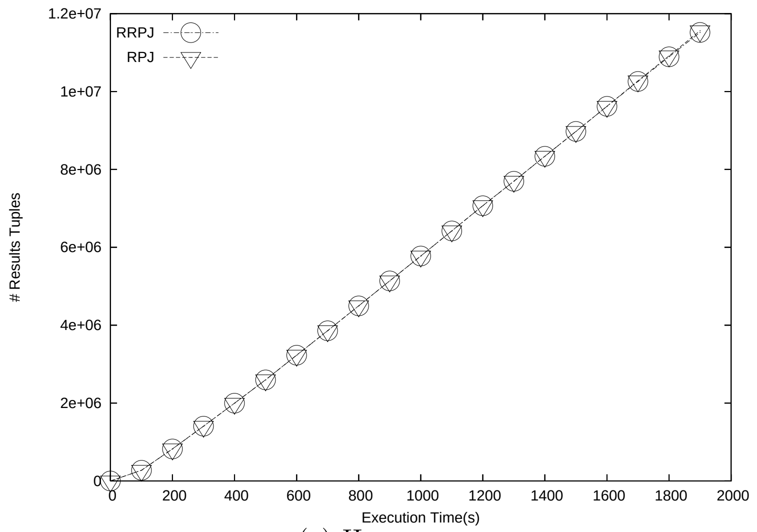
4.1 Performance Evaluation

In this section, we study the performance of the proposed *RRPJ* against *RPJ*. All the experiments were conducted on a Pentium 4 2.4GHz CPU PC (1GB RAM). We measure the progressiveness of the various flushing policies by measuring the response time.

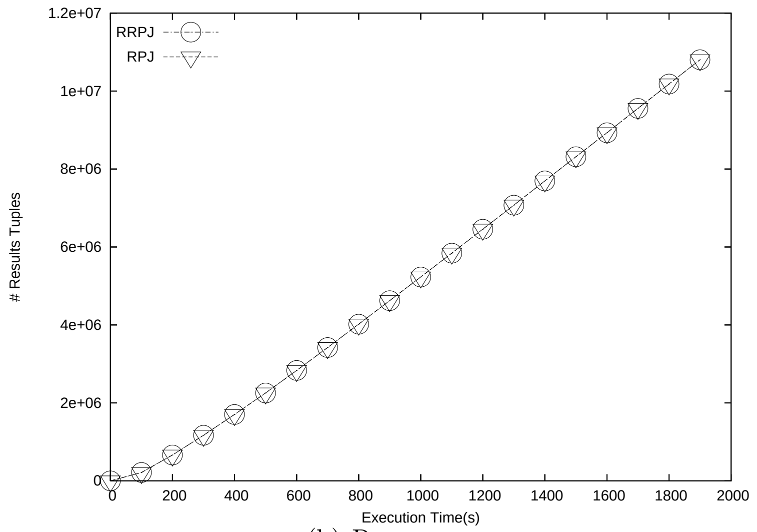
The experimental parameters are given in Table 4.1. Unless otherwise stated, the datasets used in the experiments uses the default values given in the table.

4.1.1 Effect of Uniform Data within partitions

We generated the datasets *HARMONY* and *REVERSE* based on the dataset generation techniques described in [TYP$^+$05]. We used the same arrival pattern *HARMONY* and *REVERSE*. In this experiment, we evaluate the performance of the RRPJ against RPJ. We measure the response time (x-axis) and the number of result tuples generated (y-axis). From Figure 4.1, we can observe that the performance of RRPJ is comparable to RPJ using the same datasets from [TYP$^+$05], and hence is at least as effective as RPJ for uniform data.



(a) Harmony



(b) Reverse

Figure 4.1: Effect of Uniform-Data Within Partitions

In addition, we also studied the performance of the algorithms by varying the number of tuples that are flushed whenever memory is full. Figure 4.2 shows the results that are produced due to the in-memory tuples, and Figure 4.3 shows the complete results that are produced. From Figure 4.2, we can observe that as the number of tuples that are flushed increases, the number of results produced reduces. This is because when more tuples are flushed from memory, there is less tuples that can be joined with newly arrived tuples from the corresponding data source. From Figure 4.2, we can also observe that when a single tuple is flushed, it maximizes the number of results produced during the in-memory join phase. However, it is important to note that this results in the disk-resident partitions to be badly organized. Consequently, this causes the cleanup phases which produces the complete result to take a longer time to be processed (shown in Figure 4.3(a)). Similar results are observed for the dataset Reverse. The graphs are presented in Figure 4.4 and Figure 4.5.

4.1.2 Effect of Non-uniform Data within partitions

In this experiment, we evaluate the performance of RRPJ against RPJ for non-uniform datasets. We used the same arrival pattern *HARMONY* and *REVERSE*. We restrict the domain for the join attribute for 50% of the tuples from one dataset (R1) to be in the range [1,5000] and the domain of the join attribute for 50% of the other dataset (R2) to be in the range [5001,10000]. We measure the response time (x-axis) and the number of result tuples generated (y-axis).

From Figure 4.6(a) and Figure 4.6(b), we can observe that the RRPJ outperforms RPJ by a large margin. This is because RPJ's local uniformity assumption breaks when the data within each partition is non-uniform. Comparatively, since RRPJ tracks the number of results, it is able to identify the partitions that are not producing any results, and hence avoid keeping tuples belonging to these non-productive partitions in memory.

In addition, we also studied the performance of the algorithms by varying the number of tuples that are flushed whenever memory is full. Figure 4.7 shows the results that are produced due to the in-memory tuples, and Figure 4.8 shows the complete results that are produced. Similar to the previous experiments, we can observe in Figure 4.7, the number of results produced reduces when the number of tuples that are flushed increases. In addition, we can observe in Figure 4.8(a) that if the number of tuples that are flushed each time is relatively small (e.g. one tuple), then the time taken for the cleanup phase (in order to produce the complete result set) is significantly large. This is because whenever few tuples are flushed to disk, it is appended to a corresponding disk partition. This causes the tuples that are on the disk partitions to be badly organized. As the cleanup phase uses a sort-merge join, additional work needs to be done to sort the large number of tuples before results can be produced. Another observation is that when the number of tuples flushed is 10% or more, it does not have significant impact on the cleanup phase. This is similar to the observation made in the earlier experiments on uniform data within partitions.

4.1.3 Varying Data Arrival Distribution

The datasets are generated as follows: We make use of a Zipfian distribution (with tunable parameter θ) to determine the partition for assigning a newly-arrived tuple. When $\theta = 0.0$, the data distribution is uniform (i.e. a newly-arrived tuple have equal probability of belonging to any of the partitions). When θ increases, the arrival distribution becomes more skewed (i.e. a newly-arrived tuple have higher probability to belong to specific partitions). In order to simulate a varying data arrival distribution, we re-order the partitions probabilities whenever every α tuples have arrived. The partitions are randomly re-ordered. For example, when $\theta = 2.0$, Table 4.2 shows the arrival probabilities. During the initial stage, the probability that a newly arrived tuple will belong to partition 1,2,3,4 and 5 are 0.68, 0.17, 0.08, 0.04 and 0.03 respectively. During each reorder, these probabilities for a newly arrived tuple to belong to a specific partition change.

In this experiment, we evaluate the performance of the Amortized RRPJ (ARRPJ) against RPJ and RRPJ, when the data arriving exhibits varying data arrival distribution (i.e the probability that a newly arrived tuple belongs to a partition changes). We vary the amortization factor, σ, for ARRPJ between 0.0 to 1.0. We call the corresponding algorithm ARRPJ-σ. When $\sigma = 0.0$, only the latest RRPJ values (i.e. number of results produced and size of data partition since the last flush) are used. When $\sigma = 1.0$, ARRPJ is exactly RRPJ (it computes the average of the

Arrival Probabilities, P	Initial	1st Reorder	2nd Reorder
	Partitions Assigned		
0.68	1	2	3
0.17	2	3	4
0.08	3	4	5
0.04	4	5	1
0.03	5	1	2

Table 4.2: Arrival Probabilities, $\theta = 2.0$

statistics over time).

In addition, α refers to the frequency of varying the data arrival distribution. For example, when α = 32k, it means that the data arrival distribution is changed after 32k tuples have arrived.

The results are shown in Figure 4.11(a)-(f). In addition, we summarize the throughput (i.e. number of result tuples produced over time) of each algorithm in table 4.3. In table 4.3, we can observe that an amortization factor = 0.0 need not necessarily be the best (highlighted in bold). There is a need to balance between the impact of past and current results. From Figure 4.11(a)-(e), we can observe that ARRPJ (with different amortization factor) performs much better than RRPJ. Also, when the data distribution changes frequently (e.g. Figure 4.11(f), α = 0k), the performance of RRPJ and ARRPJ are similar.

α	RRPJ	ARRPJ-0.0	ARRPJ-0.2	ARRPJ-0.5	ARRPJ-0.8	ARRPJ-1.0
0	4113	4128	4128	4125	4119	4113
4	6735	7719	7950	7665	7541	6735
8	9783	12266	12009	11503	10551	9783
16	11879	20133	20038	19428	17307	11879
20	10027	25140	**25152**	24554	20887	10027
32	12177	36388	36053	34685	27120	12177

Table 4.3: Throughput of various methods (Summary of Fig 4.11)

When α = 0k, the data arrival distribution is re-ordered aggressively (changes each time a tuple arrives). Thus, all the methods (including RPJ and XJoin) perform similarly. This is because none of the methods can make use of the statistics gathered to do effective prediction of which tuples to keep in memory combined with a generally smaller number of possible results. However, when α increases from 4k to 32k, we can observe that ARRPJ (with different α) outperforms RRPJ. This is because ARRPJ was able to better reduce the impact of the past results by amortizing the RRPJ values. RRPJ does not perform as well, since RRPJ does not differentiate between past and current results. From Figure 4.11, we can also observe that as the data changes less frequently (i.e. when α varies from 0K to 32K), the total number of result tuples significantly increases. This is because when the data distribution changes less often, the statistics computed could be used for more effective prediction of which tuples need to be kept in memory.

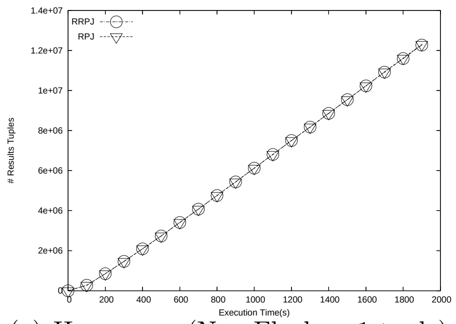
In addition, we also conducted additional experiments where we varied ρ (percentage of pages flushed each time memory is full, and θ (skewness of the data distribution). Similar trends are observed. When θ is 0.0 (i.e. uniform data), all methods (i.e. RPJ, RRPJ, ARRPJ) performs the same.

These experiments suggest however that several factors influence the correct evaluation of the output statistics when data distribution is changing over time. The amortization formula must be tuned with respect to the size of the buffer, the percentage and size of the replaced partitions as well as the frequency of the replacement. While the purpose of this paper is to introduce the idea of amortization and illustratively quantify its potential, such fine tuning is left to future work.

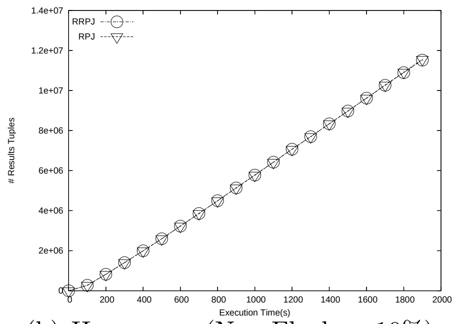
4.2 Summary

In this chapter, we presented an instantiation of the RRPJ framework for relational equijoin. Through extensive empirical studies, we show that the relational instantiation of the RRPJ framework is indeed effective and efficient. In addition, the relational instantiation do not rely on a local uniformity assumption within each hash partitions. This allows the relational instantiation to be used for skewed datasets, where the data within each partition is non-uniform.

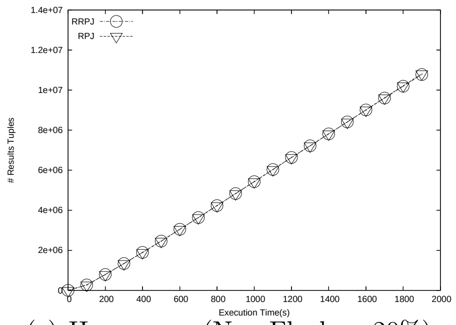
In the subsequent chapters, we will show other instantiations of the RRPJ framework for other data models. This includes the progressive processing of spatial, high-dimensional and XML data.



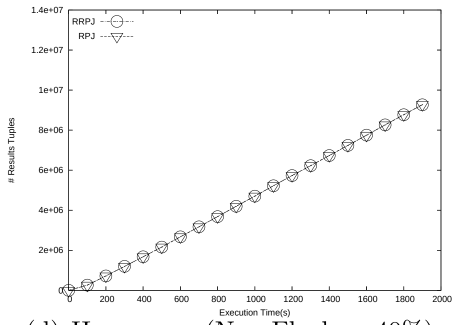
(a) Harmony - (NumFlush = 1 tuple)



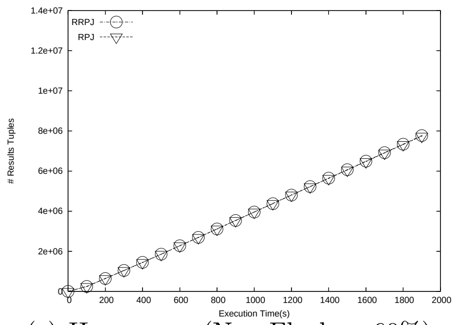
(b) Harmony - (NumFlush = 10%)



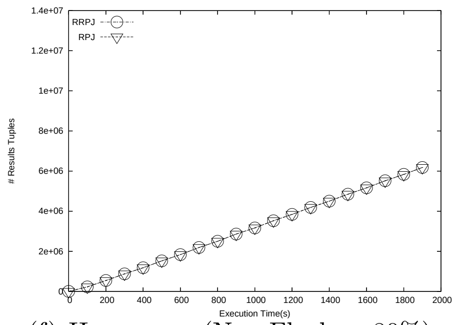
(c) Harmony - (NumFlush = 20%)



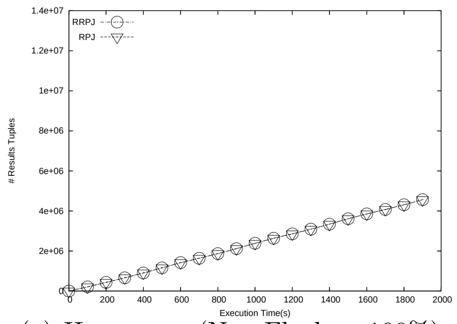
(d) Harmony - (NumFlush = 40%)



(e) Harmony - (NumFlush = 60%)

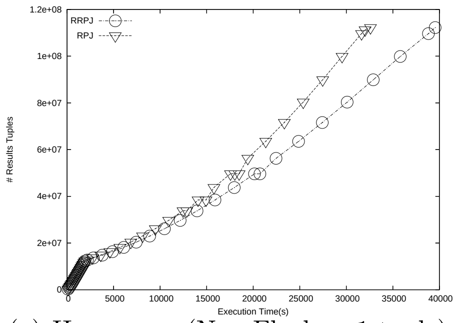


(f) Harmony - (NumFlush = 80%)

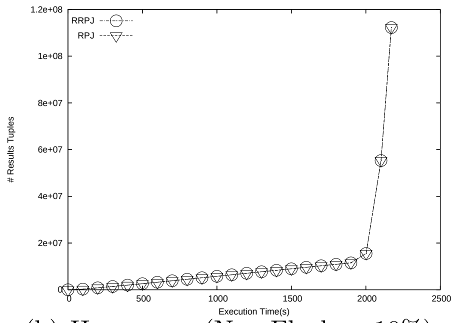


(g) Harmony - (NumFlush = 100%)

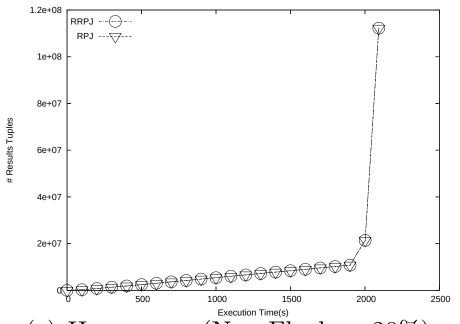
Figure 4.2: Effect of Uniform-Data Within Partitions - Harmony (Varying Number of tuples flushed)



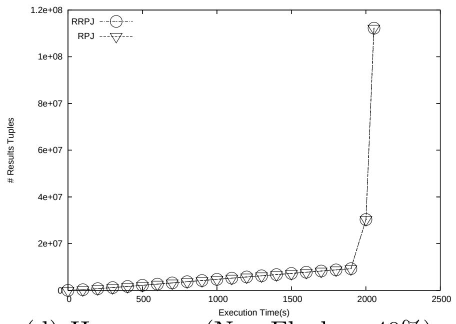
(a) Harmony - (NumFlush = 1 tuple)



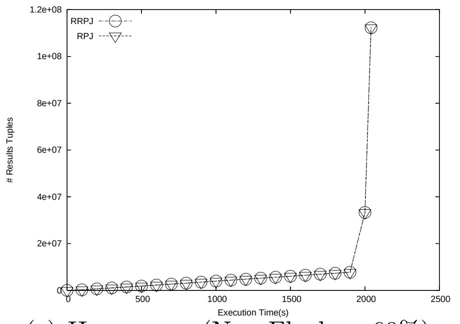
(b) Harmony - (NumFlush = 10%)



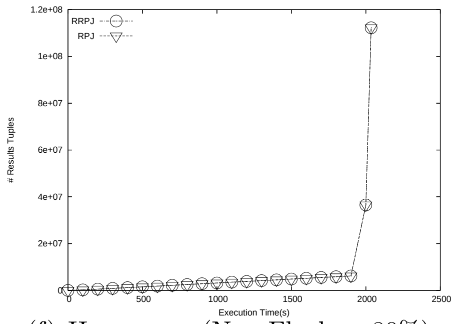
(c) Harmony - (NumFlush = 20%)



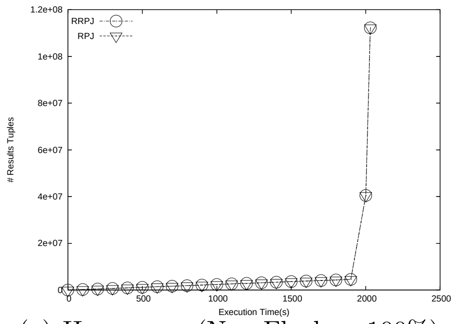
(d) Harmony - (NumFlush = 40%)



(e) Harmony - (NumFlush = 60%)

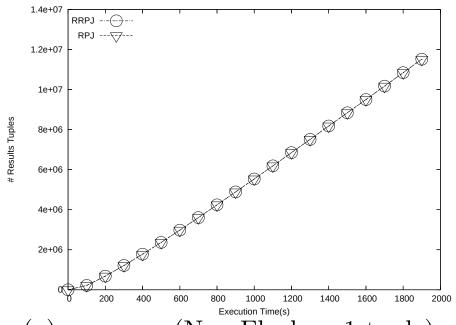


(f) Harmony - (NumFlush = 80%)

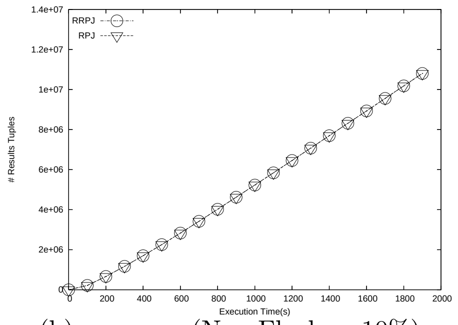


(g) Harmony - (NumFlush = 100%)

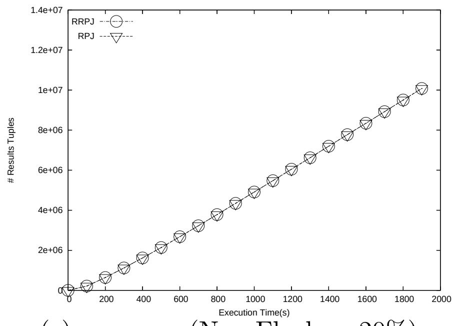
Figure 4.3: Effect of Uniform-Data Within Partitions - Harmony (Varying Number of tuples flushed / Complete results produced)



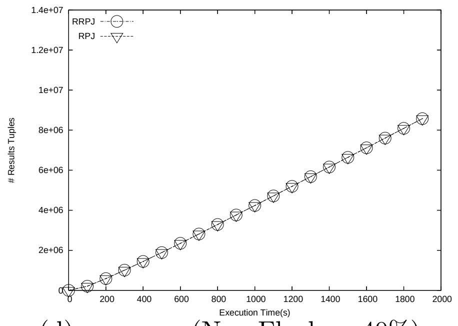
(a) reverse - (NumFlush = 1 tuple)



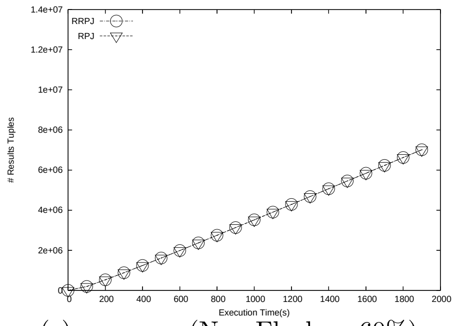
(b) reverse - (NumFlush = 10%)



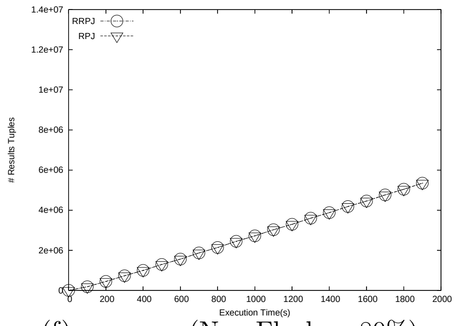
(c) reverse - (NumFlush = 20%)



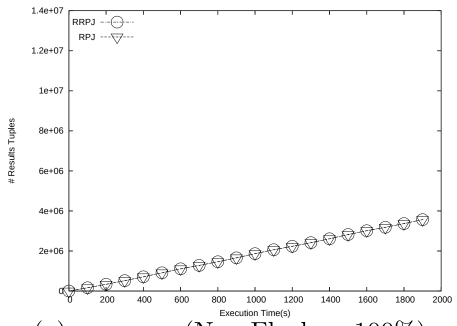
(d) reverse - (NumFlush = 40%)



(e) reverse - (NumFlush = 60%)

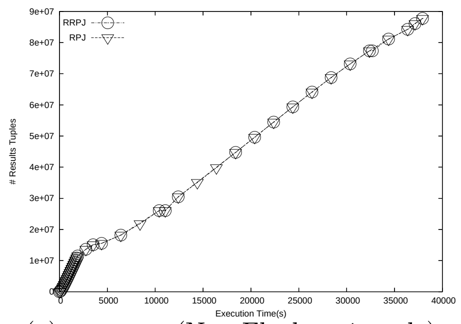


(f) reverse - (NumFlush = 80%)

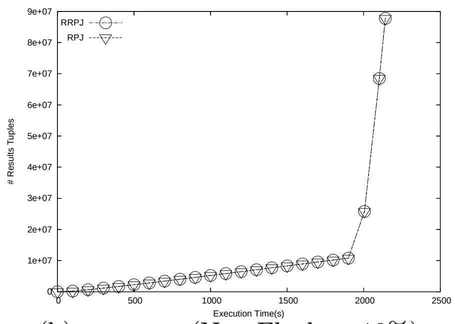


(g) reverse - (NumFlush = 100%)

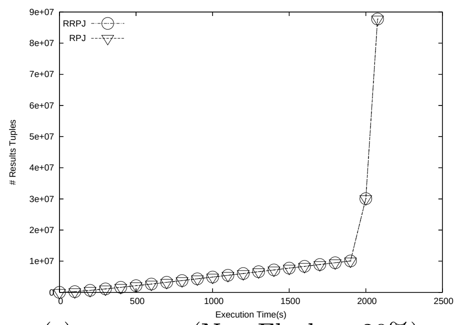
Figure 4.4: Effect of Uniform-Data Within Partitions - Reverse (Varying Number of tuples flushed)



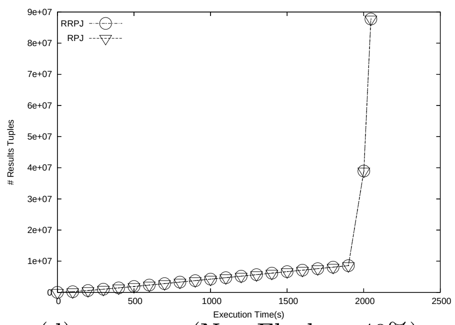
(a) reverse - (NumFlush = 1 tuple)



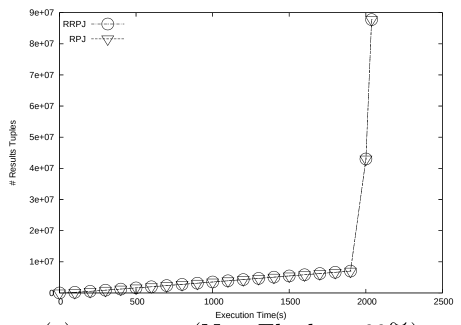
(b) reverse - (NumFlush = 10%)



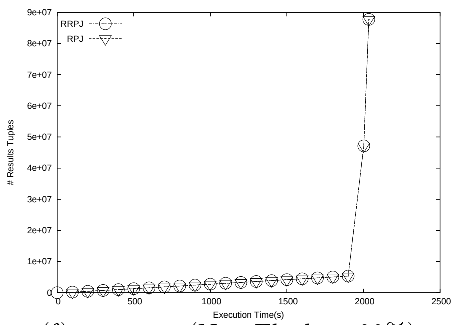
(c) reverse - (NumFlush = 20%)



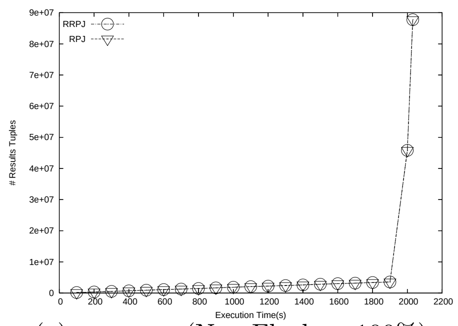
(d) reverse - (NumFlush = 40%)



(e) reverse - (NumFlush = 60%)

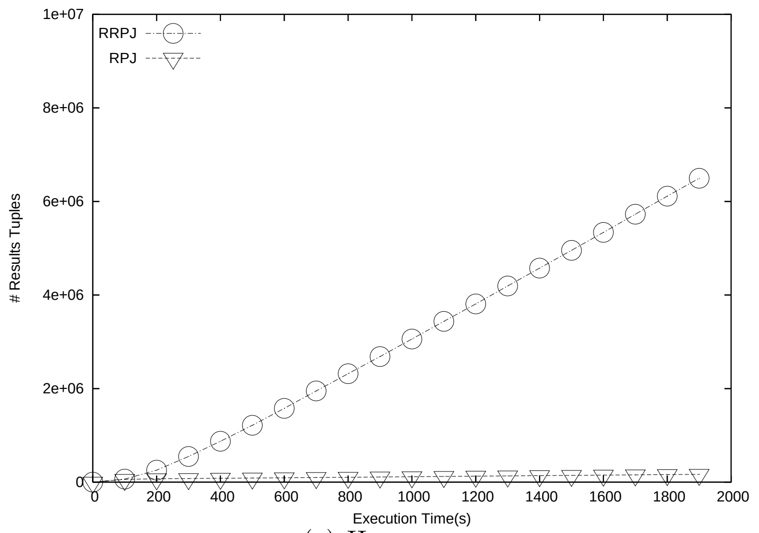


(f) reverse - (NumFlush = 80%)

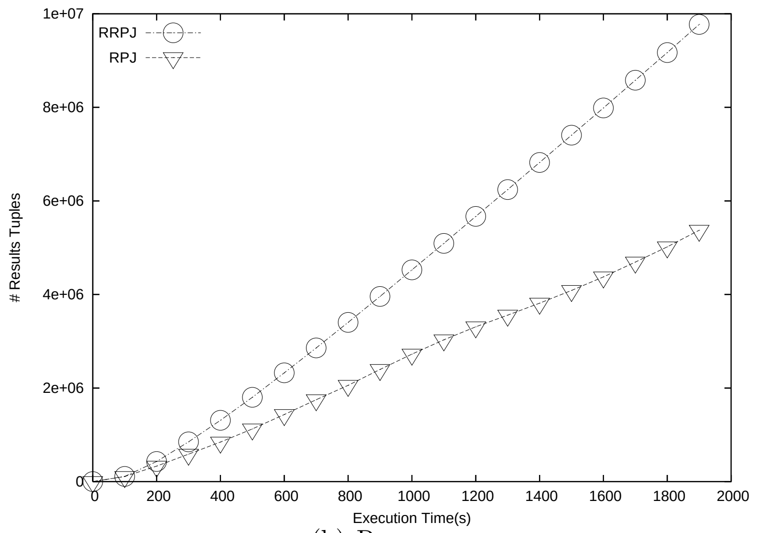


(g) reverse - (NumFlush = 100%)

Figure 4.5: Effect of Uniform-Data Within Partitions - Reverse (Varying Number of tuples flushed / Complete results produced)

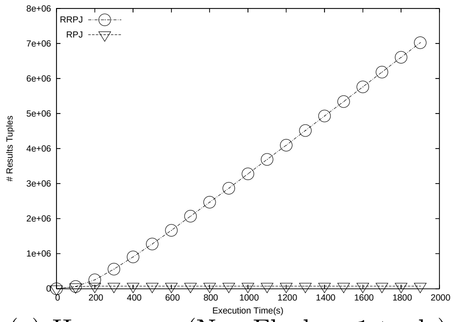


(a) Harmony

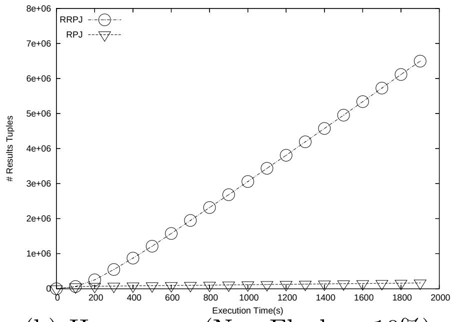


(b) Reverse

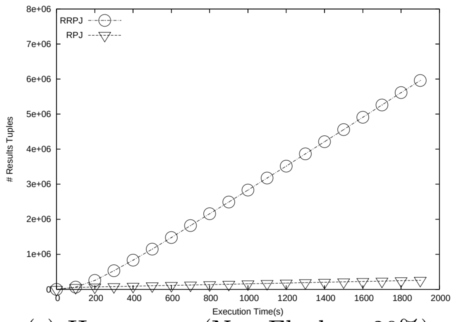
Figure 4.6: Effect of Non-Uniform-Data Within Partitions



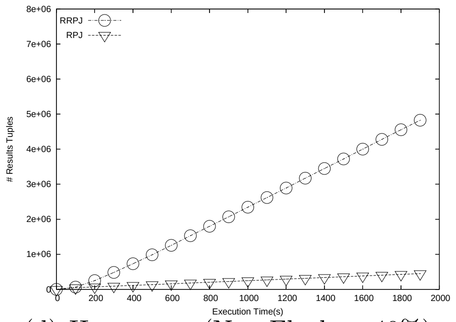
(a) Harmony - (NumFlush = 1 tuple)



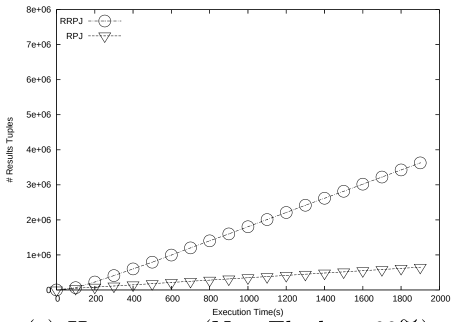
(b) Harmony - (NumFlush = 10%)



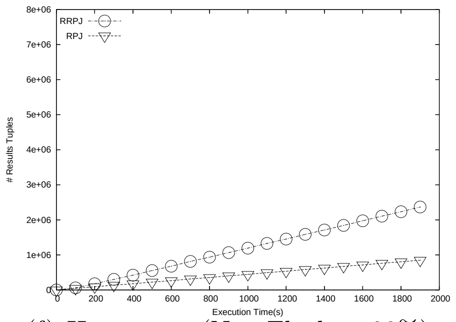
(c) Harmony - (NumFlush = 20%)



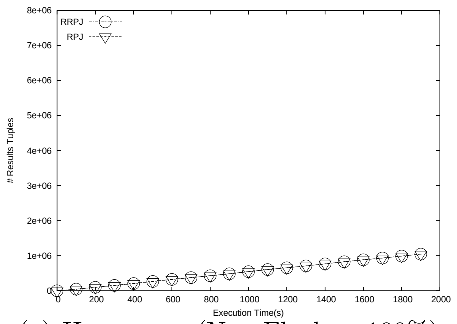
(d) Harmony - (NumFlush = 40%)



(e) Harmony - (NumFlush = 60%)

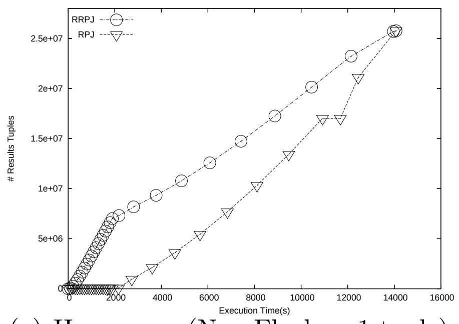


(f) Harmony - (NumFlush = 80%)

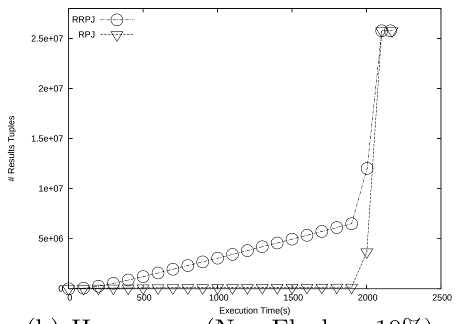


(g) Harmony - (NumFlush = 100%)

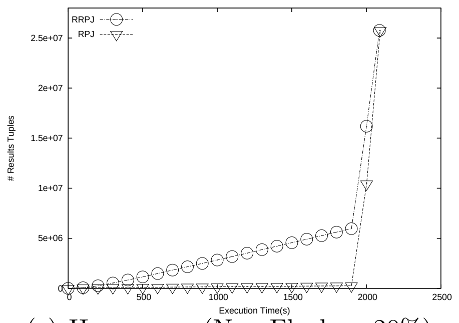
Figure 4.7: Effect of non-uniform-Data Within Partitions - Harmony (Varying Number of tuples flushed)

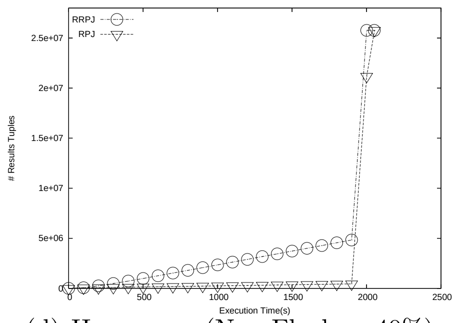


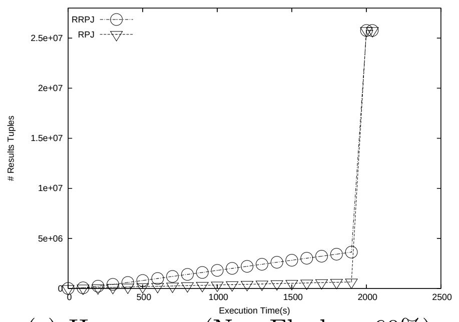
(a) Harmony - (NumFlush = 1 tuple)

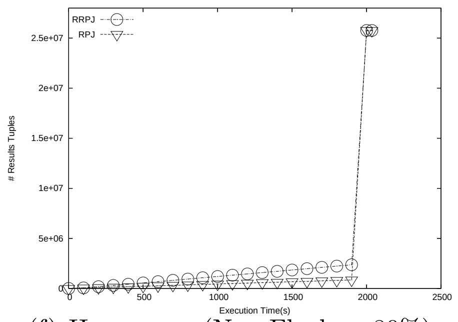


(b) Harmony - (NumFlush = 10%)


(c) Harmony - (NumFlush = 20%)


(d) Harmony - (NumFlush = 40%)


(e) Harmony - (NumFlush = 60%)


(f) Harmony - (NumFlush = 80%)

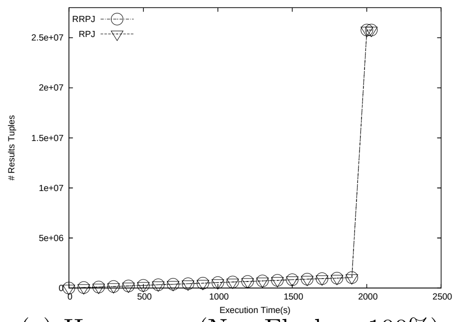
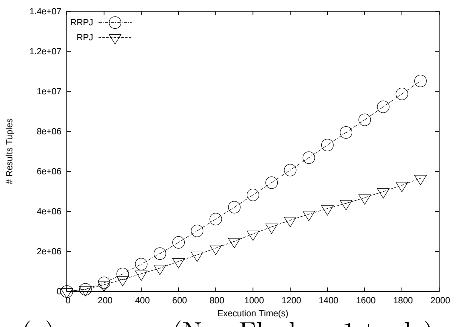
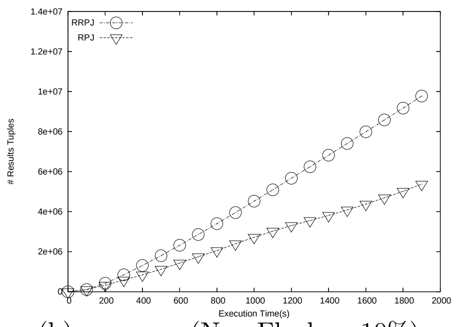

(g) Harmony - (NumFlush = 100%)

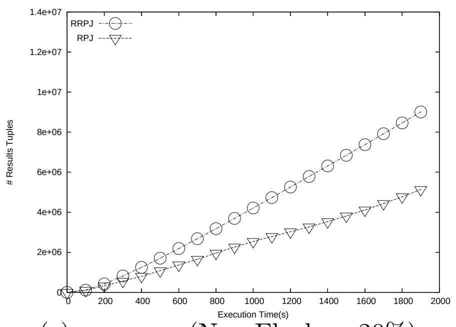
Figure 4.8: Effect of non-uniform-Data Within Partitions - Harmony (Varying Number of tuples flushed / Complete results produced)



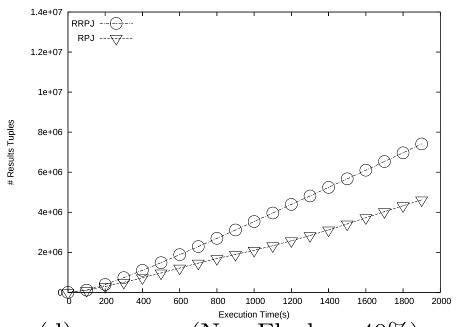
(a) reverse - (NumFlush = 1 tuple)



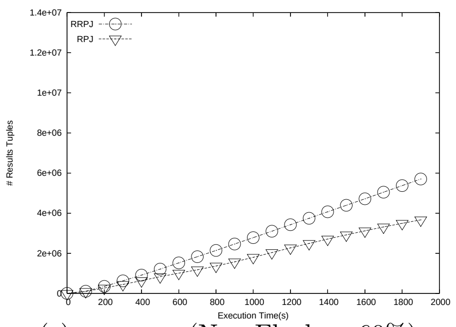
(b) reverse - (NumFlush = 10%)



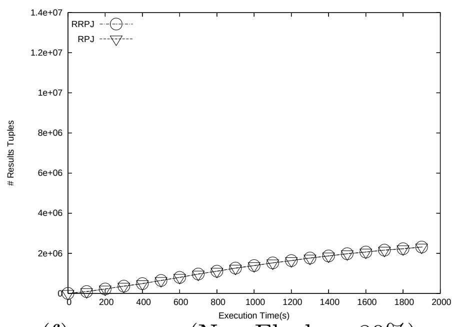
(c) reverse - (NumFlush = 20%)



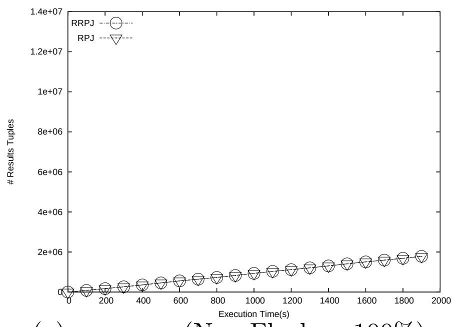
(d) reverse - (NumFlush = 40%)



(e) reverse - (NumFlush = 60%)

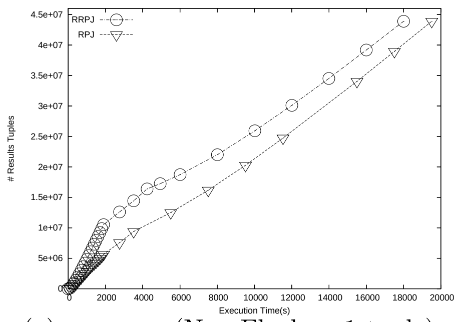


(f) reverse - (NumFlush = 80%)

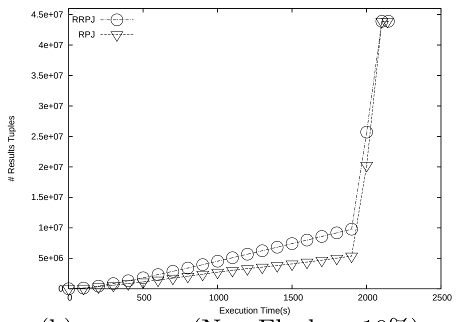


(g) reverse - (NumFlush = 100%)

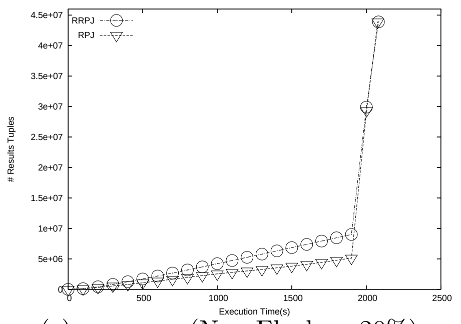
Figure 4.9: Effect of non-uniform-Data Within Partitions - Reverse (Varying Number of tuples flushed)



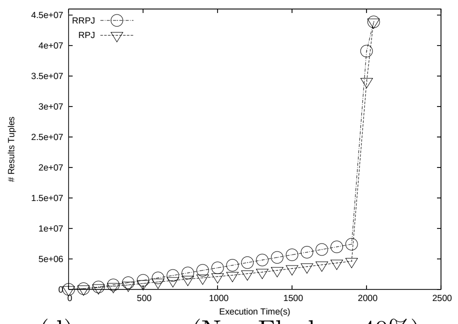
(a) reverse - (NumFlush = 1 tuple)



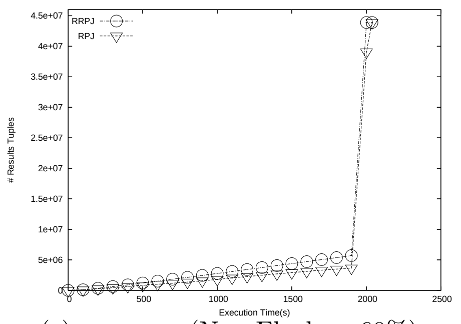
(b) reverse - (NumFlush = 10%)



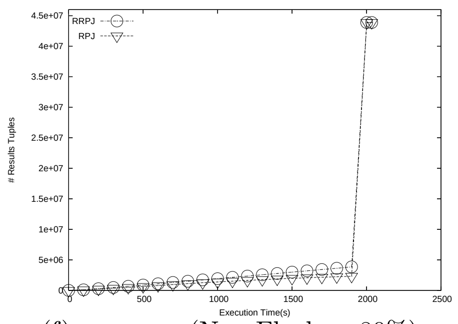
(c) reverse - (NumFlush = 20%)



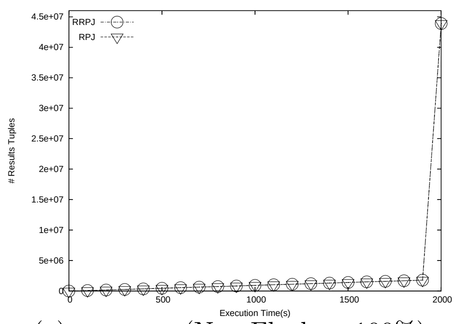
(d) reverse - (NumFlush = 40%)



(e) reverse - (NumFlush = 60%)

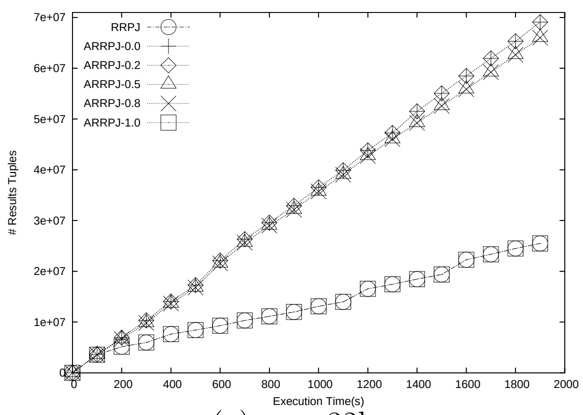


(f) reverse - (NumFlush = 80%)

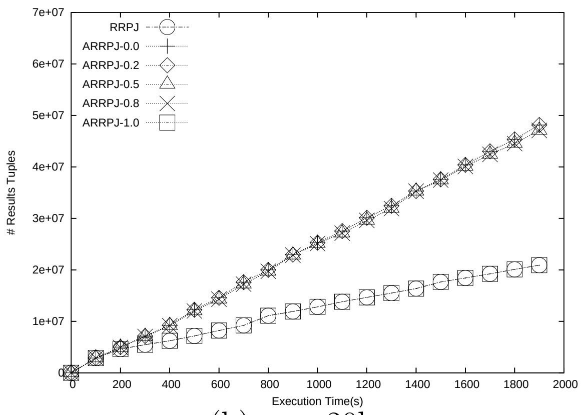


(g) reverse - (NumFlush = 100%)

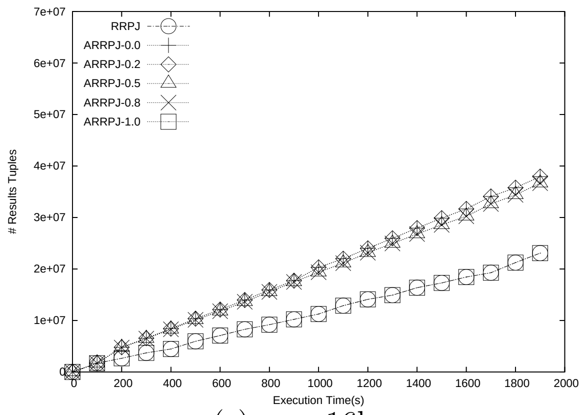
Figure 4.10: Effect of non-uniform-Data Within Partitions - Reverse (Varying Number of tuples flushed / Complete results produced)



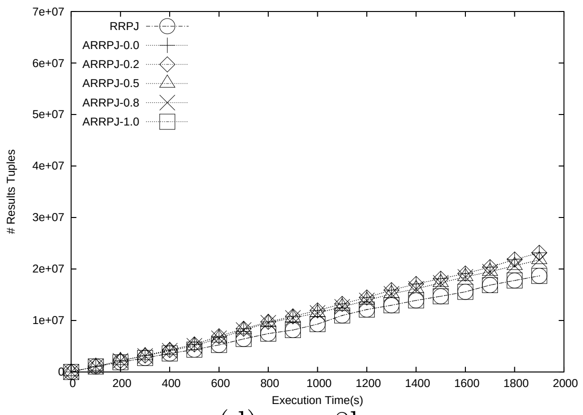
(a) α = 32k



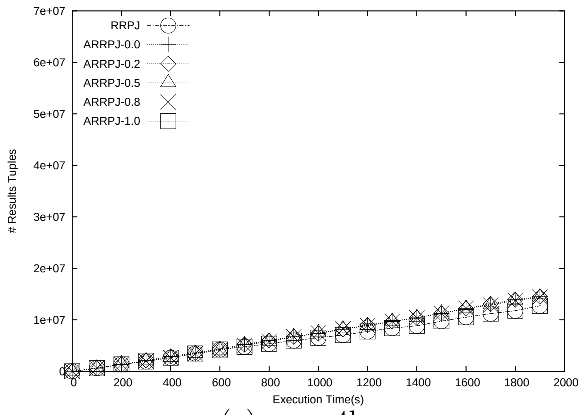
(b) α = 20k



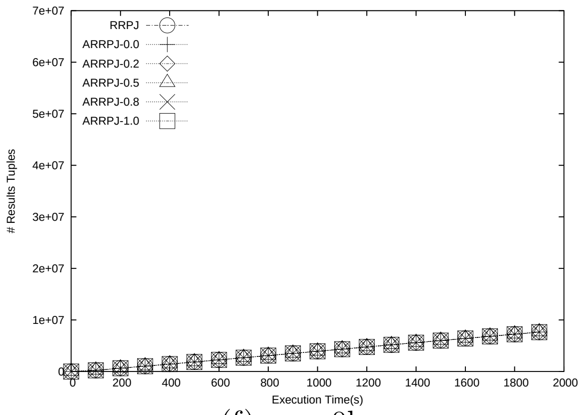
(c) α = 16k



(d) α = 8k



(e) α = 4k



(f) α = 0k

Figure 4.11: Varying Data Distribution

Chapter 5

Progressive Spatial Join

In the chapter, we present the instantiation of the RRPJ framework for the progressive spatial join using limited main memory.

The progressive spatial join problem is defined as follows: We consider spatial objects which are streamed from remote data sources through an unpredictable network. Let the two sets of spatial objects be denoted by $R = \{r_1, r_2, \ldots, r_n\}$, and $S = \{s_1, s_2, \ldots, s_m\}$, where r_i and s_j denotes the *i-th* R and *j-th* data objects respectively. In a spatial join, a result refers to a pair of objects, from each of the spatial data streams, which satisfies a spatial predicate. In this thesis, we focus on the most commonly used spatial predicate - *intersection*. Formally, (r_i, s_j) is reported as the result if r_i intersects s_j. Without loss of generality, we assume that the data objects are approximated by their respective minimum bounding rectangles (MBRs). By progressive, we refer to the ability of the join algorithm to continuously deliver results steadily as data arrives.

In addition, we assume that that the memory size is small relative to the total number of objects that needs to be joined. Our goal is to maximize the join result throughput of the spatial intersection join using the tuples that are kept in memory.

5.1 Grid-Based Progressive Spatial Join

A key requirement for an efficient progressive spatial join is that each partition should only be matched against the corresponding partition in the other data source. To partition spatial data, we can make use of either the spatial partitioning technique (similar to [LR96]) or a grid-based approach [PD96]. In [LR96], the Spatial Hash Join assumes that the data sets are local and uses the *seeded tree* for partitioning the data space. However, the *seeded tree* technique cannot be used as the basis for building a non-blocking, progressive spatial join algorithm due to its blocking nature (i.e. the seeded tree is built only when the entire dataset has arrived!).

We propose to make use of a grid-based approach, to support fast insertions and probing. Whenever a new MBR from a spatial data stream arrives, it is used to probe the grid of the other spatial data stream. The use of a grid-based approach is attractive because it reduces the need to scan all the in-memory objects, but is restricted to probing only the grid-cells in which it intersects. This greatly reduces the search space. We make use of the same flushing strategy for relational joins. The advantages of a grid-based approach are as follows. Firstly, it does not require pre-processing of the entire dataset, and hence makes it favorable for supporting online spatial join algorithms, which requires fast response. Secondly, yhe cells are a simple and natural analogy to the partitions used in a relational hash join. Similar to the pairwise comparisons of corresponding partitions (partitions with the same hash value) in a relational hash join, it suffice to compare corresponding cells, one each from the data source

In the grid-based approach, the data universe is first partitioned using an equi-width grid consisting of cells, with width w. Each grid cell consists of a memory ($cell_{ij}^{mem}$) and disk portion ($cell_{ij}^{disk}$), where i, j denotes the row and columns identifiers respectively (illustrated in Figure 5.1). Each $cell_{ij}^{disk}$ contains the MBRs that has been flushed from memory. Whenever a new MBR arrives, it is inserted into the in-memory grid cells it intersects with.

The key idea is that whenever a new data object arrives, it is hashed into one or several of the grid cells in which it intersects. The use of hashing in spatial joins was first explored in the Spatial Hash Join (SHJ) work [LR96, PD96]. SHJ assumed that no indexes are pre-constructed. In order to deal with the *coherent assignment* problem [LR96] (inherent in spatial joins), data needs to be replicated into several grid cells. The advantage of the replication is that it allows pairwise grid cells to be matched exactly once, and hence reduces the need to scan the entire grid, greatly reducing the overall computation cost. However, duplicate results are produced due to the replication and are removed in an online manner using the *Reference Point Method* in Section 5.1.1.

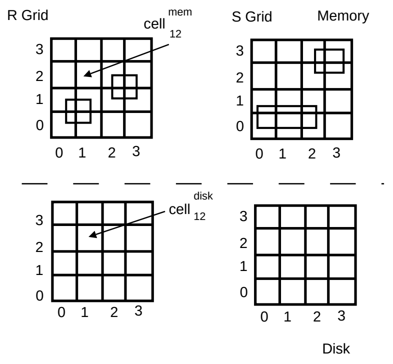


Figure 5.1: Memory and Disk Partitions

If the spatial distribution of the data objects is skewed, some of the partitions will have more data objects than the others. To ensure that the data objects are balanced uniformly amongst the partitions, the *tiling* method used in [PD96] can be used. In the *tiling* method, P partitions is created using a grid with N cells, where $N \geq P$. Each grid cell is also referred to as a *tile*. A tile-to-partition mapping is used to map the partitions to a tile. Several mapping functions are described, which includes: (1) Round-robin or (2) Hashing. The tiling method is illustrated in Figure 5.2, where $N = 9$ and $P = 4$. A round-robin tile-to-partition mapping assigns each tile to its corresponding partitions.

5.1.1 Duplicate Removal

. We make use of the Reference Point method in [DS00] for duplicate removal in the progressive grid-based spatial join. In a grid-based progressive spatial join, duplicate

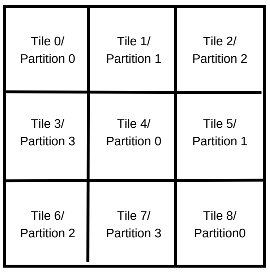


Figure 5.2: Tiling Method : Round-Robing Tile-Partitioning

results can be produced due to the following: (1) MBRs that are inserted into multiple grid cells could produce duplicate results. (2) Multiple invocation of the join between the in-memory cells with the disk-resident cells (during the *md* stages), and the disk-resident cells with the disk-resident cells (during the *dd* stages) could produce duplicate results. To ensure correctness of the spatial join algorithm, the duplicates results needs to be removed.

To tackle the first issue, the *Reference Point Method* [DS00] is used to prevent duplicate results from being generated from the insertion of an MBR to multiple grid cells. In Figure 5.3, MBR r and MBR s intersects. Since an MBR is assigned to the grid cells in which it intersects, MBR r would be inserted to the cells (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2) and (2,2), and MBR s would be inserted to the cells (1,1), (2,1), (3,1), (1,2),(2,2),(3,2), (1,3), (2,3) and (3,3). During the spatial join, when the MBRs in pairwise grid cells are joined, results would be produced four times since MBR r and s joins in cells (1,1), (2,1), (1,2) and (2,2). In order to do duplicate elimination, the *Reference Point* method uses a *point*, p, to determine which grid cell the result should be generated. Given that an MBR is defined by the coordinates $(x1, y1, x2, y2)$, the reference point, p, between two MBRs r and s is defined as follows:

$$p = (max(r.x1, s.x1), min(r.y2, s.y2))$$

When computing the spatial join between pairs of corresponding grid cell, the result is generated if the reference join of the two intersecting MBRs falls in the same grid cell. For example, in Figure 5.3, the result is only generated once in the cell (1,2), indicated by the reference point (i.e. black dot).

To tackle the second issue, we make use of the timestamp technique described in *XJoin* [UF99]. Each MBR is associated with two timestamps: *Arrival timestamp* (ATS) and *Departure Timestamp* (DTS). ATS is set when the MBR arrives from its data source, whereas DTS is set when the MBR is flushed to a disk partition. Once assigned, the timestamp cannot be modified. If the MBR is in memory, its DTS is set to be ∞. The interval between ATS and DTS denotes the time in which the MBR is in memory. If the {ATS,DTS} timestamps of a pair of MBRs (one from each of the data source) overlaps, it means the results have already been produced during the time in which both MBRs were in memory. Hence, when the same pair is compared during the *md*-stage, no results needs to be produced. In the *md*-stage, results are produced for pairs of MBRs whose {ATS,DTS} timestamp do not overlap.

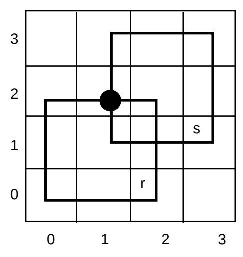


Figure 5.3: Reference Point Method

5.1.2 Flushing Strategy Variants

In this section, we discuss various different strategies that is used to identify the MBRs that are flushed to disk whenever memory is full. We propose a naive extension to RPJ for spatial data, which we call RPJ Spatial.

Naive extension to RPJ for spatial data (RPJ) A naive extension to the RPJ [TYP+05] model is that instead of using 1D partitions (for relational data), we use 2D partitions (i.e. grid). RPJ estimates the probability P(R_1) and P(R_2) by maintaining a counter n_i^{rcnt} for each relation R_i (initially set to the number of arriving R_i tuples between the initial time interval [0,1]). In the *RPJ Spatial* model, the arrival probability $p_i^{arr}(v)$ of a tuple belonging to relation R_i and belongs to the partition (j, k) is then computed in Equation 5.1 (Refer to [TYP+05] for the complete proof for the 1D partition model). In *RPJ Spatial*, we maintain counters $n_i^{total}[(j,k)]$ for each Cell(j,k) (i.e. cell at row j, column i in a grid) for a r x c grid.

$$P_i^{arr}[Cell(j,k)] = \frac{n_i^{total}[Cell(j,k)]}{\sum_{j=1}^{r}\sum_{k=1}^{c} n_i^{total}[Cell(j,k)]} \cdot \frac{n_i^{rcnt}}{n_1^{rcnt}+n_2^{rcnt}} \tag{5.1}$$

When memory is full, *numFlush* data objects residing in the cells with the smallest $P_i^{arr}[Cell(j,k)]$ are flushed to disk.

RRPJ In RRPJ, we track the number of results and the size of each partition and uses Equation 3.3 to compute the Th_i values in order to determine the tuples to be flushed to disk.

RRPJ-F In RRPJ-F, we track the number of results produced by each tuple (using Equation 3.3), and use this to determine which tuples to be flushed to disk. Since the results are tracked per tuple, the size of a partition is set to 1.

5.2 Performance Evaluation

In this section, we evaluate the performance of the various flushing strategy against a naive extension to RPJ. The algorithms are implemented in C++. All the experiments were conducted on a Pentium 4 2.4GHz CPU PC (1 GB RAM). The memory/disk page size is fixed at 4096 bytes. We measure the progressiveness of the various flushing strategies by considering the number of data that have arrived vs the number of results produced. Without loss of generality, we use MBR representation for data objects.

In all experiments, we assume that there are two finite spatial data streams, R and S. The parameters and values for the experiments are presented in Table 5.1.

5.2.1 Dataset Generation

The spatial datasets used in the experiments are generated as follows. We divide the space into a n x m grid, with equi-width grid cells. Each grid cell has width w. For each data stream, we randomly pick a (i,j) cell (i-row, j-column in the grid, i $\leq$ n, j

Parameter	Values
Disk Page Size	4096 bytes
Grid Size	10 rows x 10 columns
Memory Size, M	1000 pages
Number of MBRs per disk page	102
Number of MBRs flushed to disk	10% of M
α, Degree of Uniformity within the grid cells	[0.2-0.8]
ρ, Degree of Replication	[0.0-1.0]
β, Number of cells which a MBR is replicated	2 (default) ,4,8
Dataset Size	2 million MBRs

Table 5.1: Experiment Parameters and Values

$\leq$ m). Next, we generated a MBR by randomly picking 4 coordinates $(x1, y1, x2, y2)$, where $(x1, y1)$ refers to the lower-left corner of the MBR, and $(x2, y2)$ refers to the upper-right corner of the MBR. We consider the data space to be $[0, 1]$ x $[0, 1]$, and the width of each cell for a 10 x 10 grid is 0.1.

Given two spatial data streams, R and S. In order to generate data (in which we vary the data uniformity for a cell), we first determine the set of αnm cells where the MBR within each cell do not intersect. Whenever we generate a (i, j) cell, and the (i, j) cell belongs to the set of the αnm cells, we divide the cells into 2 equal sub-cells. A R MBR is generated for one of the sub-cell and a S MBR is generated for the other sub-cell.

In order to generate data for varying ρ, we follow the same data generation procedure. Next, we randomly generate a number, *dprob* ($0 \leq dprob \leq 1$). If $dprob \geq \rho$, then we extend the MBR size to overlap with the number of cells specified by β.

5.2.2 RPJ vs RRPJ

In this experiment, we evaluate the performance of methods which uses statistical models on the data distribution (RPJ) against our proposed model of using the result-distribution (RRPJ).

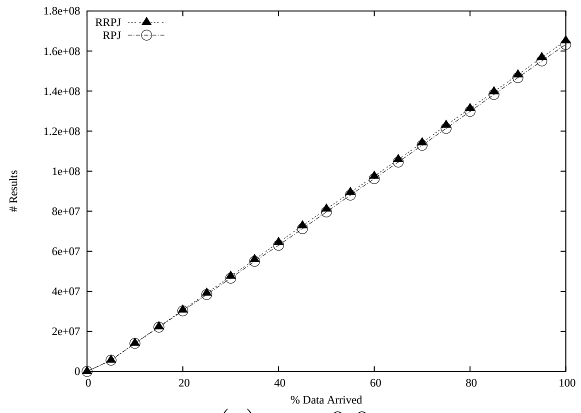
We vary the effect of data uniformity within the grid cells. The parameter, α, determines the percentage of cells in the grid where the MBR belonging to two data streams R and S do not overlap. When $\alpha = 0.0$, the MBRs within each cell is uniformly distributed (i.e. MBRs from R and S have equal chances of intersecting). When $\alpha = 1.0$, the MBRs from R and S do not overlap (i.e. no join results is produced). We vary α from 0.0 to 0.8 (1.0 is omitted because no results is produced). We measure the number of results produced (y-axis) vs the percentage of the data that has arrived (x-axis).

From Figure 5.4(a), we can observe that when $\alpha = 0.0$, the performance of all the flushing strategies are similar. When α varies from 0.2 to 0.8 (Figure 5.4(b)-(c)), we can observe that *RRPJ* outperforms all the other methods. The main reason is because since *RRPJ* tracks the result-distribution, it is able to distinguish between productive cells, containing large number of MBRs, (which produce results) compared to un-productive cells containing large number of MBRs (which do not produce results). Comparatively, the spatial version of *RPJ* was not able to perform as well due to the assumption of local uniformity within each partition/cell.

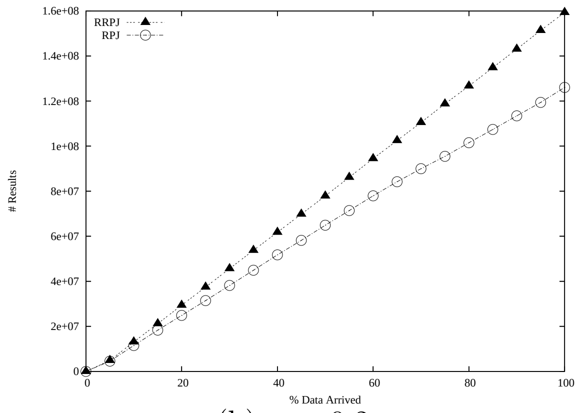
5.2.3 Effect of Spatial Extents

In this section, we study the effect of replication of the ids, ρ, on the performance on *RRPJ* and *RRPJ-f*. We fix β= 2 (i.e. if a MBR is duplicated, it occupies 2 grid cells). We vary the probability in which a MBR will be replicated. This is denoted by ρ which varies from 0.0 to 1.0. When $\rho = 0.0$, then the MBR are not replicated, and each MBR generated from the respective data streams fit into a grid-cell. When $\rho = 1.0$, then every MBR that arrives needs to be duplicated into 2 grid cells. We measure the number of results produced (y-axis) with respect to the percentage of data that has arrived (x-axis).

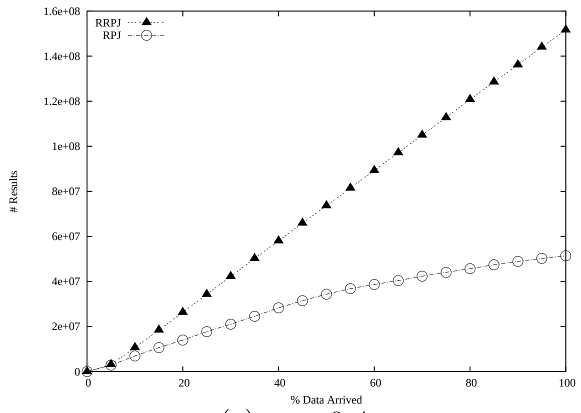
From Figure 5.5, we can observe that *RRPJ-f* outperforms *RRPJ* by producing more results. This is because *RRPJ-f* tracks the individual results produced by each tuple, and hence it able to more accurately determine the tuples which will produce more results. This incurs more space compared with *RRPJ* which tracks only the



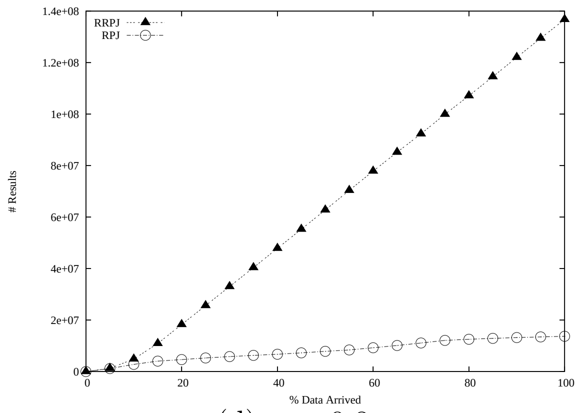
(a) $\alpha = 0.0$



(b) $\alpha = 0.2$



(c) $\alpha = 0.4$



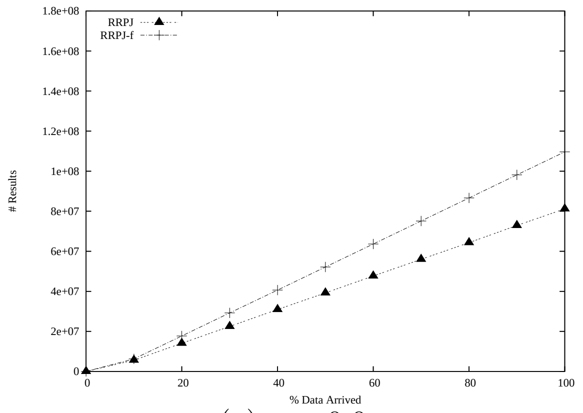
(d) $\alpha = 0.8$

Figure 5.4: Varying Data Uniformity within Grid

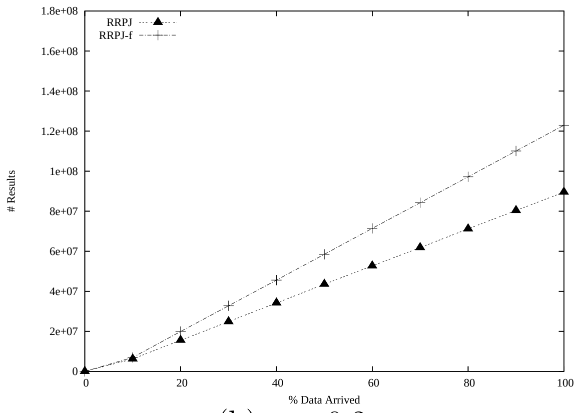
total results produced by each partition.

5.3 Summary

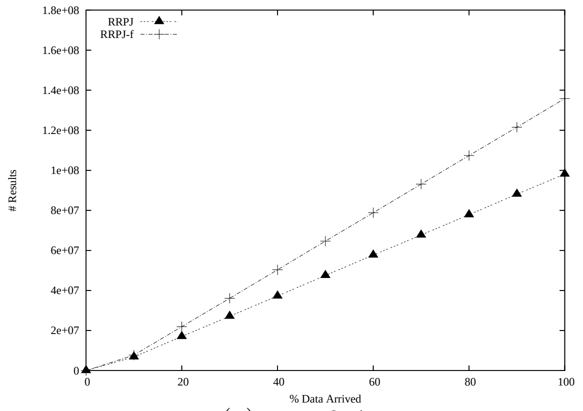
In this chapter, we presented an instantiation of the RRPJ framework for the spatial intersection join. This further emphasis the generic nature of the flushing policy that is introduced in the RRPJ framework, which allows it to be instantiated for other data models easily.



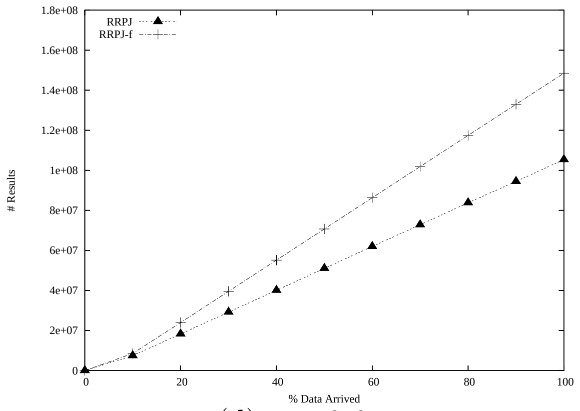
(a) $\rho = 0.0$



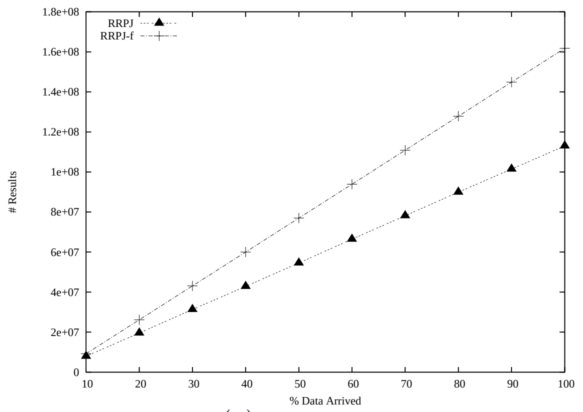
(b) $\rho = 0.2$



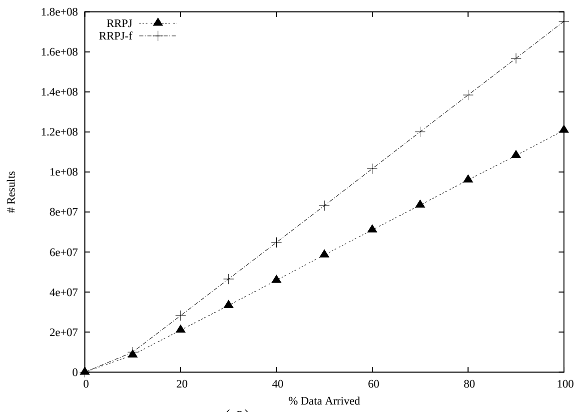
(c) $\rho = 0.4$



(d) $\rho = 0.6$



(e) $\rho = 0.8$



(f) $\rho = 1.0$

Figure 5.5: Varying Degree of Replication

Chapter 6

Progressive Distance Similarity Join

Conventional distance similarity join algorithms batch process datasets that reside on local storage. The algorithms are blocking. They are unsuitable for progressively computing the similarity join of streams of high-dimensional data as they cannot produce results progressively, i.e. as soon as data is available.

In this chapter, extending the RRPJ principle, we propose an effective and efficient algorithm for the progressive computation of the similarity of high-dimensional data that are streamed from remote data sources, using limited main memory. We consider two d-dimensional bounded data streams R and S. We refer to data from R and S as R_i and S_j respectively ($0 \leq$ i $\leq |R|$, $0 \leq$ j $\leq |S|$), where $|R|$ and $|S|$ are the total number of data objects in R and S respectively. Each data point consists of d values. Given a data point R_i, the values are $(r_{i1}, r_{i2}, \ldots, r_{id})$, where r_{ix} denotes the x-th value ($1 \leq$ x $\leq$ d). Similarly, for a data point S_j, the values are $(s_{j1}, s_{j2}, \ldots, s_{jd})$.

The results of a similarity join between R and S, *SimJoin(R,S)*, consists of all object pairs (R_i, S_j), where $D_d(R_i, S_j) \leq \epsilon$, Here, we consider without loss of generality D_d to be the Euclidean distance, where $D_d(R_i, S_j) = (\sum_{x=1}^{d} |(r_{ix} - s_{jx})^2|)^{\frac{1}{2}}$. ϵ is a user-defined threshold, which determines the maximum dis-similarity between R_i and S_j. Notice that the similarity join is symmetrical.

6.1 Grid-Based Similarity Join

We use the probe-and-insert approach as described in [WA91] and [TBL07c].

6.1.1 Probing

Whenever a new tuple, t_d, arrives (from one of the data streams), it is used to probe the in-memory tuples from the other data stream. In order to efficiently identify the tuples to be probed, a d-dimensional grid is used to partition the data space. The scanning for potential result tuples is restricted to the cell in which t_d falls into and to its neighboring grid cells (those within ϵ distance of the border of the grid cell).

We first identify the grid cell in which t_d falls into and the cells that are within ϵ-distance. Once the cells are identified, we check whether each tuple, t, in the grid can be joined by checking the Euclidean distance between t_d and t.

We keep track of the number of results produced by each grid cell using a counter, $numResults$. Once the probing of the grid cell c is completed, we update the statistics for the grid cell.

6.1.2 Insertion and Flushing

We then identify the grid cell in which the new tuple should be inserted (Line 1). If there is space, t_d is inserted into its own grid. If memory is full, we invoke $FlushDataToDisk()$ which flushes data to disk to make space for newly arrived tuples. We then insert t_d into the grid cell g (Line 4).

For each ith cell of the grid (with 1 ≤ i ≤ n, where n is the total number of grid cells), we maintain a count. The cells to be flushed are determined based on this value. The two flushing strategies that we propose differ in the way the value is computed (described in Section 6.1.3) and the partitions to be flushed are selected. Partitions are flushed until $NumFlush$ (user-defined) tuples have been flushed.

6.1.3 Flushing Strategies

Naive Extension to RPJ (neRPJ)

We propose an extension to RPJ, called Naive Extension to RPJ (neRPJ) for high-dimensional data. The neRPJ algorithms maintains the $neRPJ$ value, that is the number of data in a cell divided by the total number of data. The opposite cell (that is the matching cell in the other streams partition) to the grid cell with the smallest $neRPJ$ value is flushed.

In the relational case, the mapping of a cell in one stream to the opposite stream is 1-to-1. When we probe for result tuples, we probe only a single cell from the opposite data stream. However, when dealing with high-dimensional data, besides probing the corresponding cell from the opposite data stream, we need to probe the neighboring cells (those within ϵ distance) as well. When neRPJ flushes an opposite cell, it might have inadvertently flushed a cell that could produce results at a later time.

Result Rate-based Flushing (RRPJ)

The Th_i value is an estimate of the productivity of the i-th cell (with 1 $\leq$ i $\leq$ n, where n is the total number of cells used to store the data). In the equation below, R_i is the total number of results produced by the i-th cell and N_i is the total number of tuples in the i-th cell.

$$Th_i = \frac{Ri}{N_i} \tag{6.1}$$

The RRPJ algorithm maintains the Th_i value (Equation 3.3). In RRPJ, the grid cells with the smallest values are flushed.

6.2 Performance Evaluation

In this section, we compare the performance of the algorithms (RRPJ, neRPJ and *Random*). We measure the number of result tuples generated (y-axis) vs percentage of data that have arrived (x-axis). In all experiments, we assume that there are two finite d-dimensional datasets. Each dataset is characterized by the data distribution and the order of arrival of the data. In Section 6.2.1, we use a uniform and skewed datasets. For the skewed dataset, we also consider various correlations between the data distributions - Harmony and Reverse [TYP$^+$05]. In addition, we compare the performance of the algorithms in two extreme cases. In the first case (Section 6.2.2), we use a 'checkered' dataset. In the second case (Section 6.2.3), we consider the case where the data in some of the grid cells are non-uniformly distributed. In Section 6.2.4, we validate the effectiveness of the proposed algorithm for real-life data using the COREL [htt99] dataset.

We implemented all the flushing strategies in C++, and conduct the experiments on a Pentium 4 2.4 Ghz PC (1GB RAM). Unless otherwise stated, the parameters presented in Table 6.1 are used for the experiments. Similar to [TBL07c], we refer to the proposed result rate-based method for high-dimensional data as the Result-Rated Based Progressive Join (RRPJ). In addition, we also included a Random method as a baseline. Whenever memory is full, the Random method randomly selects a grid cell to be flushed to disk.

Table 6.1: Experiment Parameters and Values

Parameter	Values
Disk Page Size	4096 bytes
Number of cells Per Dimension	4
Memory Size, M	1000 pages
Number of points per disk page	85
Number of MBRs flushed to disk	10% of M
Dataset Size (for 2 streams)	500K data points
Similarity Join Distance Threshold, ϵ	0.1, 0.2, 0.3

6.2.1 Uniform and Skewed Dataset

The goal of these experiments is to compare the performance of the algorithms using uniform and skewed datasets. In addition, we also vary the order of arrival of the data. In Figure 6.1, we can observe that the results are consistent with earlier observations for relational data [TBL07c]. When the data from the datasets are uniformly distributed, each tuple is equally probable to contribute to a result. Hence, all flushing strategies are equally efficient and as good as random. This is illustrated in Figure 6.1.

In the next experiment, we consider skewed dataset. We simulate clustered data by dividing the space into a d-dimensional grid, and by varying the cardinality of the grid cells based on a Zipfian distribution. We set the skewed factor for the Zipfian distribution, σ to be 1.0. Thus, some grid cells have more data than others. In addition, we also investigated the impact of the correlation between the two data streams on the datasets (we use two schemes used in [TYP+05] called *HARMONY* and *REVERSE*). In the *HARMONY* scheme, corresponding clusters on each stream have the same density of data. In the *REVERSE* dataset, corresponding clusters have reverse densities (according to the grid numbering). In addition, we use a third scheme in which data is reverse and arrive in a random order.

RRPJ outperforms the other methods in all cases. It is more the case with a *REVERSE* randomized dataset (Figure 6.4a-c) than with a *REVERSE* Figure (6.3a-c), than again with a *HARMONY* dataset (Figure 6.2a-c). In other words, RRPJ is capable of adapting to the irregularities of the datasets distribution and arrival.

6.2.2 Checkered Data

We now consider the extreme case in which data is generated by alternating the cells in which the data falls into on each stream. In one dataset, only the even cells contain data. In the other dataset, only the odd cells contain data. Thus, the data in the two data streams are somehow 'disjoint'. We refer to the dataset as the checkered dataset.

From Figure 6.5, we can see that RRPJ outperforms neRPJ. This is because whenever memory is full, neRPJ first determines the cells with the lowest neRPJ values and flushes the cells in the other data stream. However, this might not be the optimal decision, since the cell that is flushed could be a cell that could contribute to a large number of results. Recall that in a high-dimensional similarity join, we do not just scan the corresponding cell, but also its immediate neighborhood. Since RRPJ determines the results for each cell, and flushes cells with the lowest Th_i values (and not the cell from the other data stream), it is able to differentiate between cells that contribute to large number of results from cells that do not.

6.2.3 Non-Uniform Data within Cells

The worst-case scenario for RPJ is when the local uniformity assumption for cells does not hold. We construct such a data set by having cells where the majority of the data in one cell do not entirely 'join' with the data in the other cell. We refer to this as non-uniformity within cells. We restrict the range of values for some of the dimensions, which we refer to as *non-uniform dimensions*. For each non-uniform dimension, we limit the random values generated to be in the range [0,0.5] for one dataset, and [0.6,1.0] for the corresponding data set. Given a d-dimensional dataset, we set d/2 of the dimensions to be non-uniform dimensions, and the remaining to be uniform dimensions. The results are presented in Figure 6.6, where we observed that RRPJ performs much better than neRPJ. This is because neRPJ relies on a local uniformity assumption for the data within cells, which does not entirely hold in this worst-case scenario. In contrast, RRPJ do not suffer from this problem because it tracks the statistics on the result output of cells. In Figure 6.6(c), we make use of ϵ = 0.3 in order to produce readable figure, but verified that the result for various ϵ values are consistent.

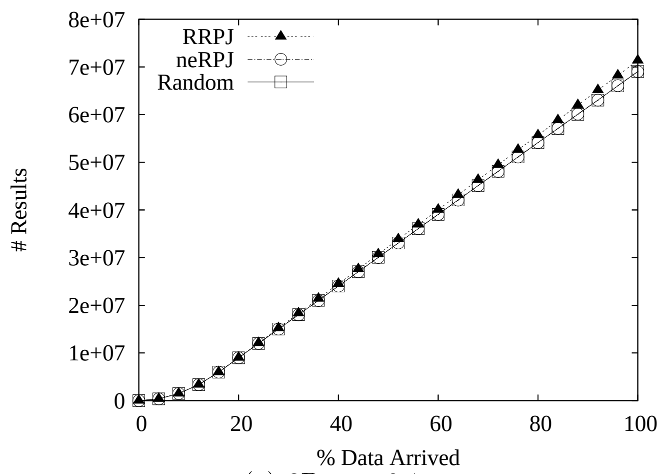
6.2.4 Real-life Datasets

Finally, we validate the effectiveness of the proposed RRPJ algorithm for real-life datasets. In this experiment, we use the Corel (Color Moment) dataset [htt99]. The Corel dataset consists of 9 dimensional features for 68,040 images. We created two data streams by randomizing the order of the data for both datasets. We then perform a self-join on the data. From Figure 6.7, we can observe that RRPJ outperforms neRPJ and Random in all cases for varying ϵ. This further reinforces the advantages from using a result-rate based approach.

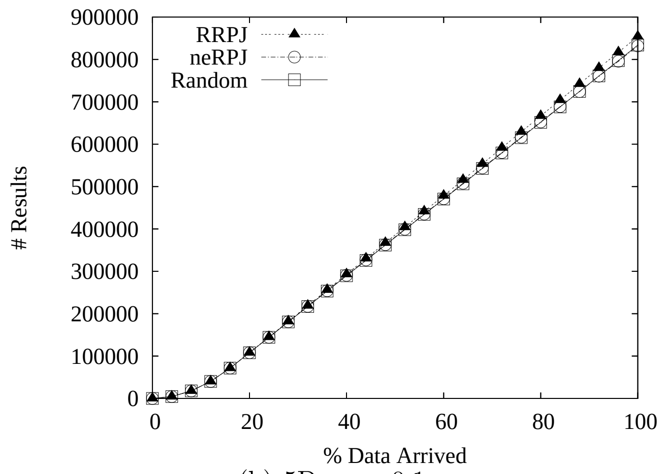
6.3 Summary

In this chapter, we propose a novel progressive high-dimensional similarity join algorithm. The algorithm uses a result-rate based flushing strategy. It is an extension of our previous work on progressive relational equijoin [TBL07c] to the case of high-dimensional data.

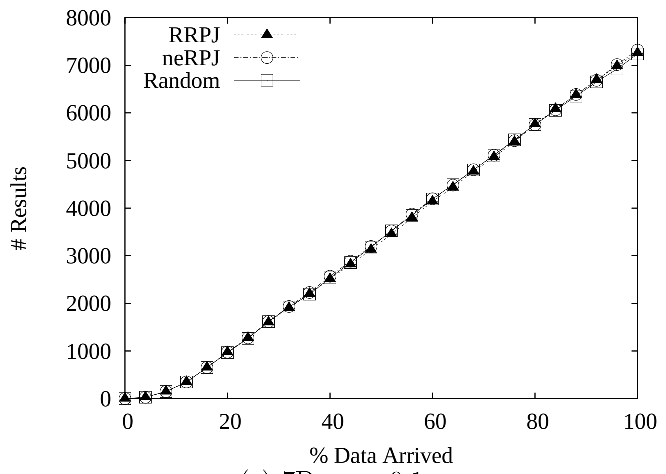
We have conducted an extensive performance analysis, comparing our proposed algorithm with a naive extension of RPJ [TYP$^+$05] (a state-of-the-art progressive relational join), called neRPJ, to high-dimensional data . Using both synthetic and real-life datasets, we have shown that our proposed method, RRPJ, outperforms neRPJ by a large margin and is therefore both effective and efficient. In contrast to conventional similarity join algorithms, RRPJ can deliver results progressively and maintain a high result throughput.



(a) 3D, $\epsilon = 0.1$

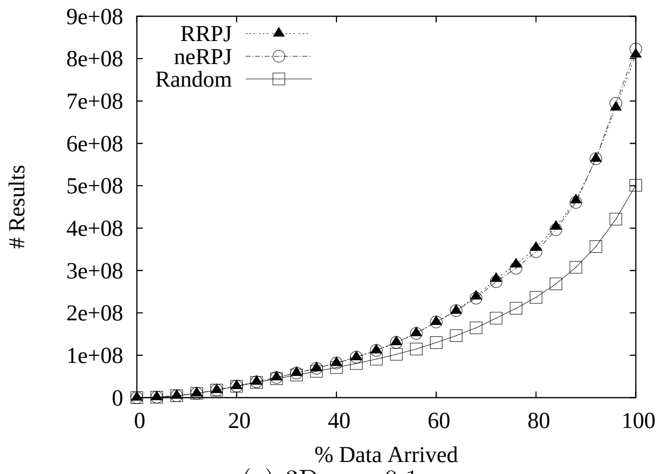


(b) 5D , $\epsilon = 0.1$

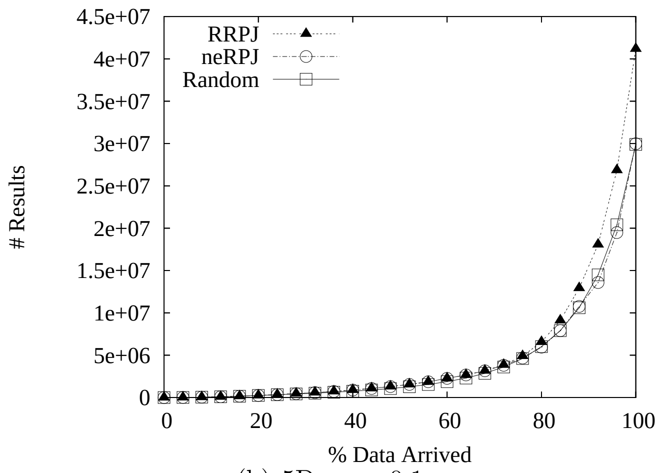


(c) 7D , $\epsilon = 0.1$

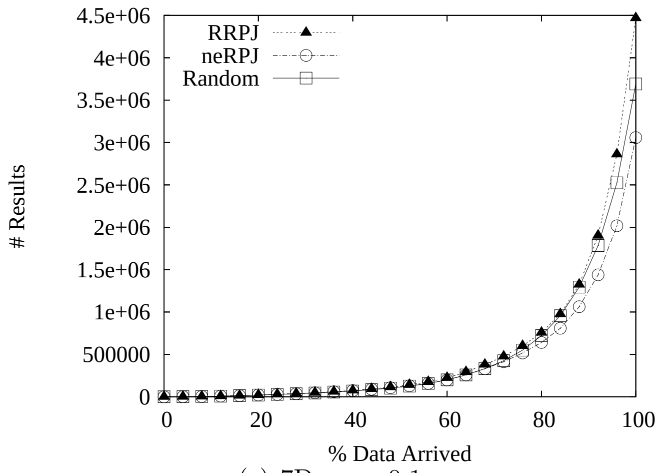
Figure 6.1: Varying Dimension: Uniform Dataset



(a) 3D, $\epsilon = 0.1$

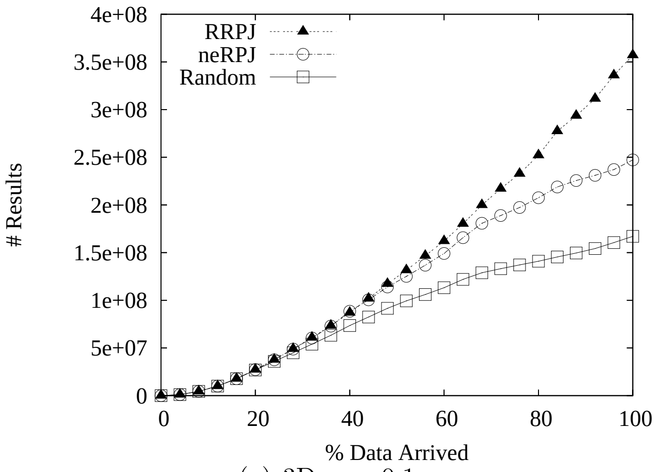


(b) 5D , $\epsilon = 0.1$

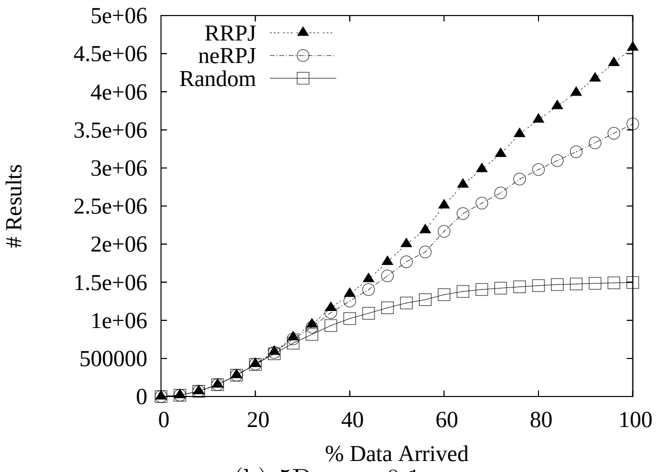


(c) 7D , $\epsilon = 0.1$

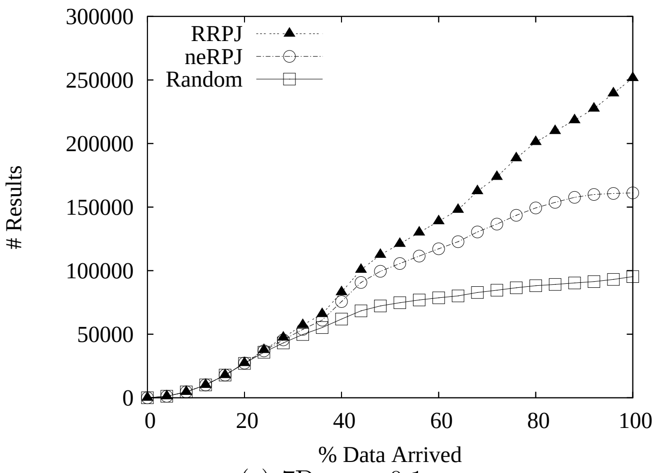
Figure 6.2: Varying Dimension: Skewed Dataset - Harmony



(a) 3D, $\epsilon = 0.1$

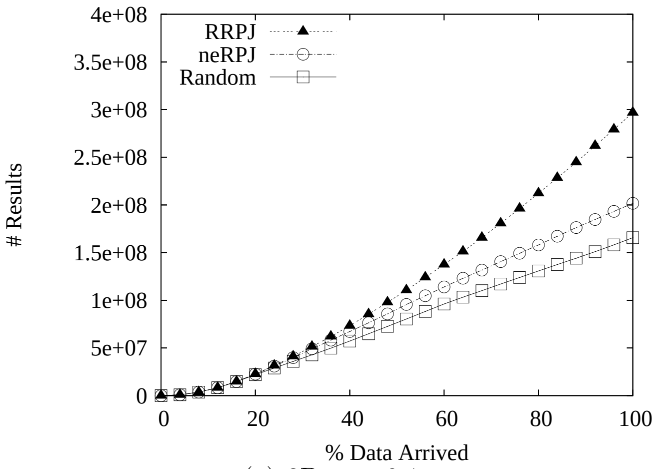


(b) 5D , $\epsilon = 0.1$

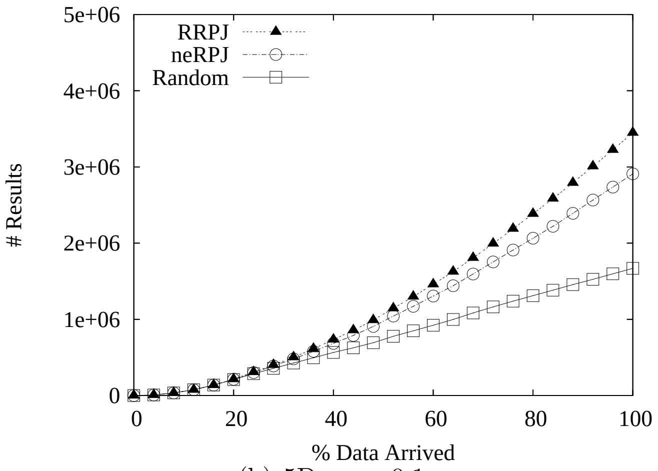


(c) 7D , $\epsilon = 0.1$

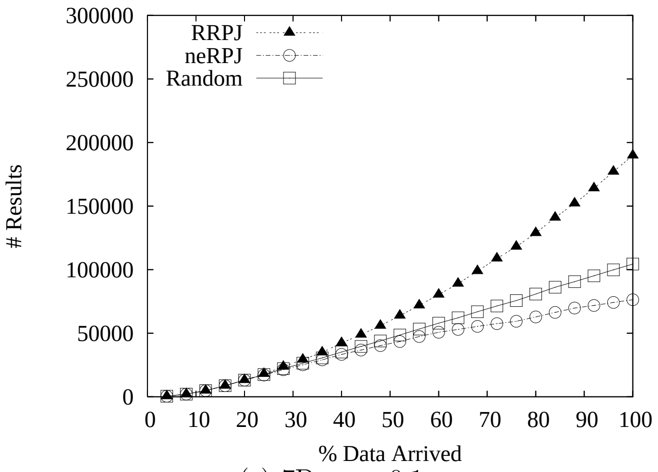
Figure 6.3: Varying Dimension: Skewed Dataset - Reverse



(a) 3D, $\epsilon = 0.1$

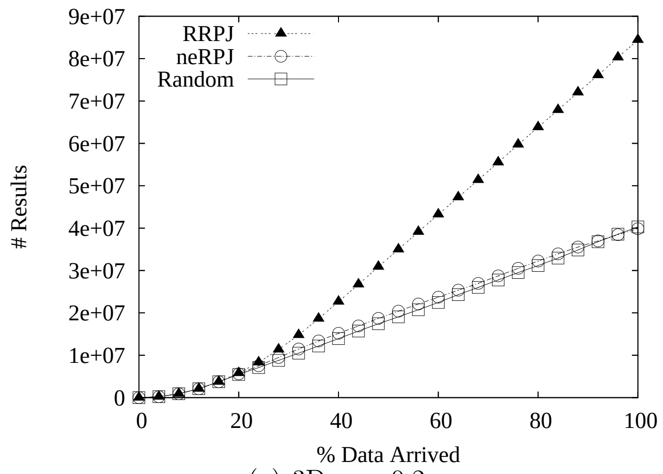


(b) 5D , $\epsilon = 0.1$

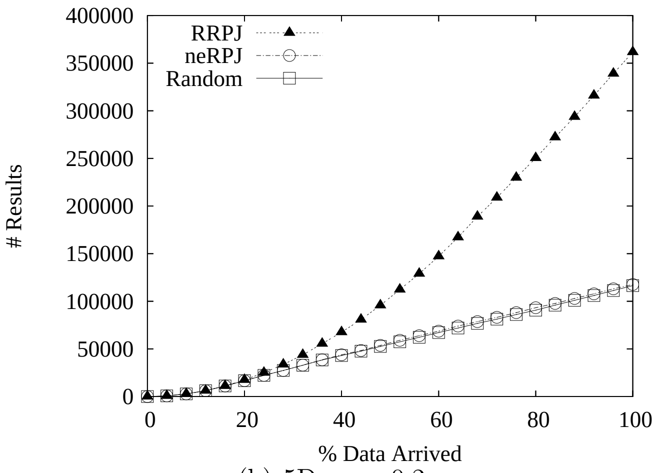


(c) 7D , $\epsilon = 0.1$

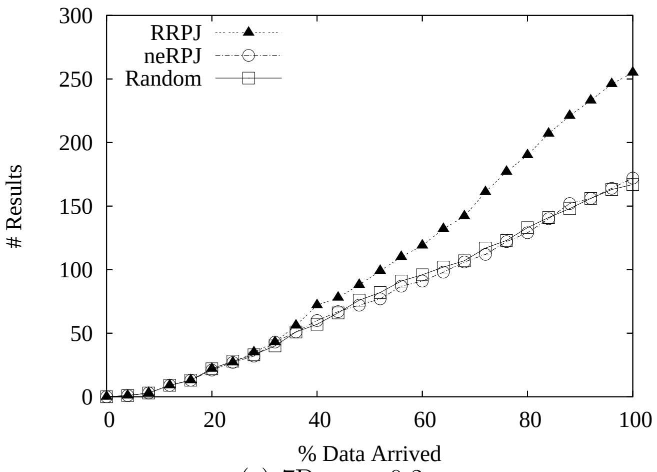
Figure 6.4: Varying Dimension: Skewed Dataset - Reverse (Randomize arrival)



(a) 3D, $\epsilon = 0.2$

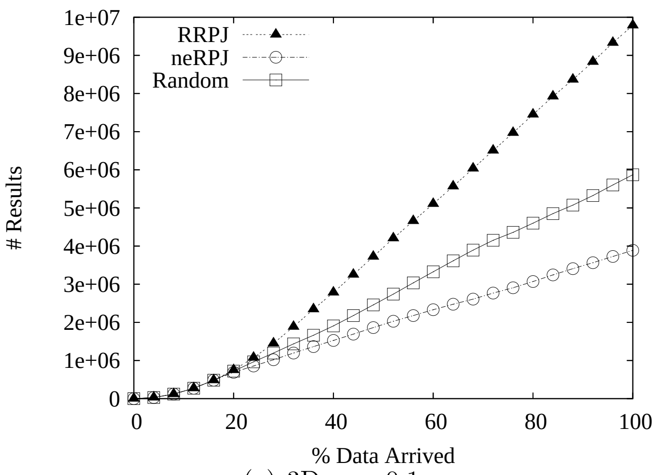


(b) 5D , $\epsilon = 0.2$

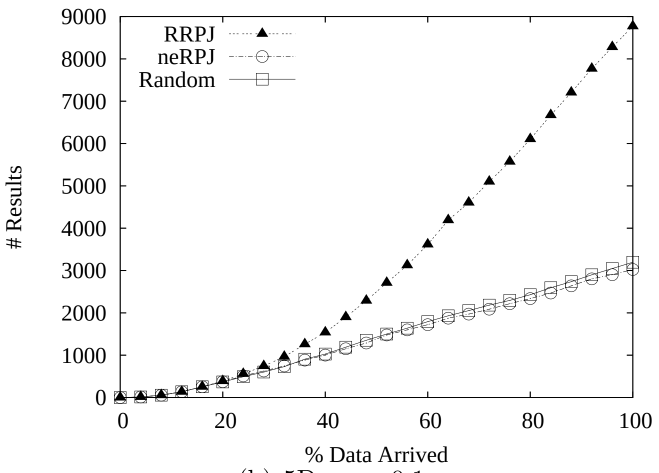


(c) 7D , $\epsilon = 0.2$

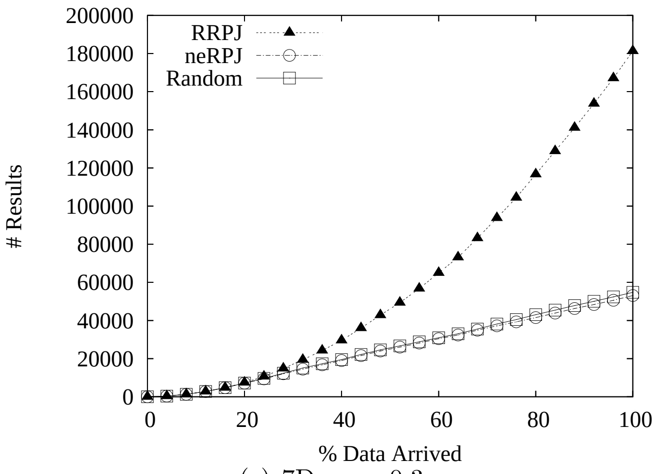
Figure 6.5: Varying Dimension: Checkered Dataset



(a) 3D, $\epsilon = 0.1$

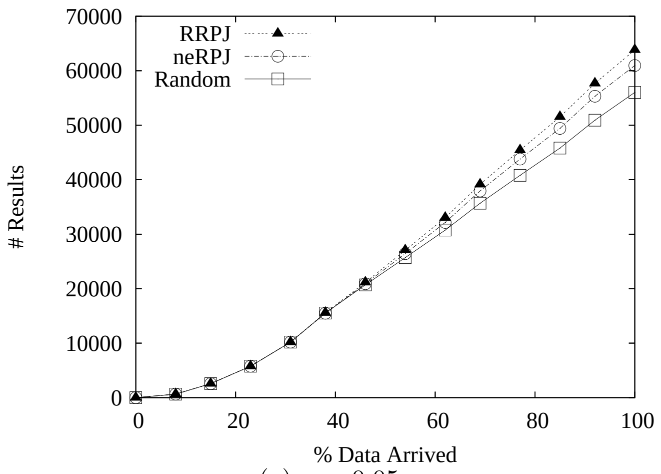


(b) 5D , $\epsilon = 0.1$

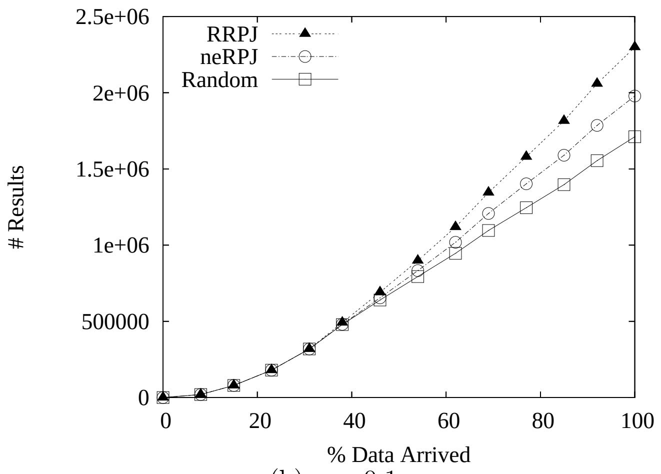


(c) 7D , $\epsilon = 0.3$

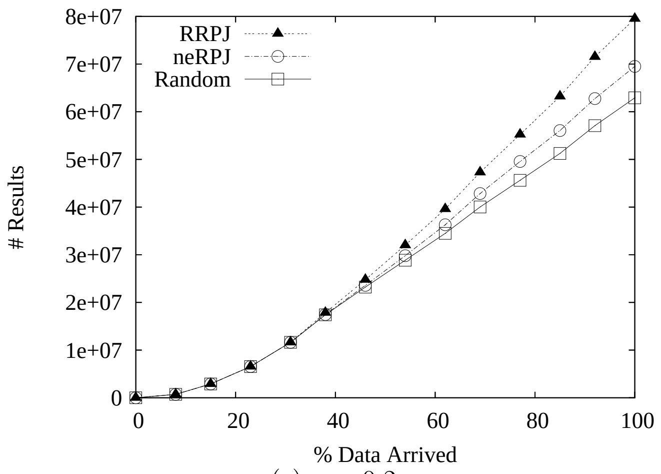
Figure 6.6: Varying Dimension: Non-Uniform Data Within Cells



(a) $\epsilon = 0.05$



(b) $\epsilon = 0.1$



(c) $\epsilon = 0.2$

Figure 6.7: Varying ϵ: COREL Dataset, 9D

Chapter 7

Progressive Join of Multiple XML Streams

In this chapter, we present an instantiation of the RRPJ framework for progressive XML processing. In addition, we also show how the result-oriented focus of the RRPJ framework can be used effectively for determining an ideal probe sequence for multi-way joins.

The ubiquity of network accessible XML data necessitates the design of XML query processors which can process complex queries over multiple XML data streams. For example, expressive RSS aggregators e.g. Yahoo Pipes [Yah07], Danaides [TBL07a]) require support for effective and efficient processing of complex queries. Thus, we need to devise XML query processors for XML languages such as XPath or XQuery that supports the processing of structural and predicate constraints as well as join queries [HDG^{+}07] over multiple XML data streams. In order to ensure a good user experience, the XML query processors must deliver initial results quickly, and maintain a consistent high result throughput. Main memory is limited and when it is full, data needs to be flushed to disk. As we need to produce results progressively with a high throughput, we need to effectively manage the XML data that is kept in memory and favor data that is most likely to contribute to the result. A key insight is to make use of statistics from either the input (i.e. data) or output (i.e. result) distributions.

We propose a practical approach to the progressive processing of (FWR) XQuery queries on multiple XML streams, called Twig'n Join (or TnJ). The query is decomposed into a query plan combining several twig queries on the individual streams, followed by a multi-way join and a final twig query. The processing is itself accordingly decomposed into three pipelined stages progressively producing streams of XML fragments. Twig'n Join combines the advantages of the recently proposed TwigM algorithm and our previous work on relational result-rate based progressive joins. In addition, we introduce a novel dynamic probing technique, called Result-Oriented Probing (ROP), which determines an optimal probing sequence for the multi-way join. This significantly reduces the amount of redundant probing for results. We comparatively evaluate the performance of Twig'n Join using both synthetic and real-life data from standard XML query processing benchmarks. We show that Twig'n Join is indeed effective and efficient for processing multiple XML streams.

The problem is defined as follows. Given two XML data streams, R and S, where the XML data are delivered tag by tag from remote data sources. twig pattern (extracted from the XQuery query) T_r and T_s are defined for R and S respectively. XML result fragments F_r and F_s are produced for portions of the XML documents that matches T_r and T_s respectively. The user define a set of join attributes A in which the XML fragments can be joined. A result $<F_r, F_s>$ is reported if F_r and F_s fulfill the join attribute condition defined by A. Our goal is to be able to progressively deliver the result.

Consider the following query example. A user is interested to know the latest news based on his blog entries. This is achieved by comparing the tag of the blog entries and the keyword for the news entries. Both the news and the blog entries are made available as RSS feeds (i.e. XML streams). In order to combine the entries from the blog and the news entries, we can make use of a join between the new and blog XML streams. The join predicate A is *q/tag =s//techNews/keyword*. This can be expressed as the following XQuery query.

```
(for $s in doc("news.xml")//item
 for $q in doc("blogs.xml")//entry
 where $q/tag=$s//techNews/keyword
   and contains($s/title, "CNA")
return
  <resultTuple>
    {($s/blurb), ($s/article), ($q/entryId) }
  </resultTuple>)
```

7.1 Twig'n Join (TnJ)

A FWR XQuery query can be decomposed into three parts: (1) Structural filtering on the input streams (2) Predicate Processing and (3) Structural filtering on the results. We assume that a XQuery pre-processor will parse the (FWR) XQuery expression and generate a query plan. During predicate processing, we can perform value-based filtering as well as process the joins between the input streams. We focus on join processing. Figure 7.1 shows a possible query plan for Scenario B (Appendix B). We note that further optimization of the query plan is possible. However, we consider query optimization as an orthogonal issue.

The query plan consists of several twig queries on the individual XML streams, followed by predicate processing and a final twig query. XML data (news.xml and blogs.xml) are continuously streamed from remote sites. The data is then matched using the two twig matching operators (TM_A and TM_B). The output from TM_A and TM_B (XML fragments) are then joined using a join operator (i.e. predicate processing).

In this thesis, we use the state-of-art TwigM machine [CDZ06] to efficiently perform the twig matches on the streaming XML data, and a hash-based join for joining the data. When intermediate XML fragments are continuously produced by the twig matching operators, the memory might become full. Whenever memory is full, we will need to flush some of these XML fragments to disk so that they can be joined at a later stage, or whenever both data streams block. In this thesis, we focus on maximizing the results from the XML fragments that are retained in memory. We make use of a Result-Rate based approach [TBL06, TBL07c] to determine the results to be flushed to disk whenever memory is full.

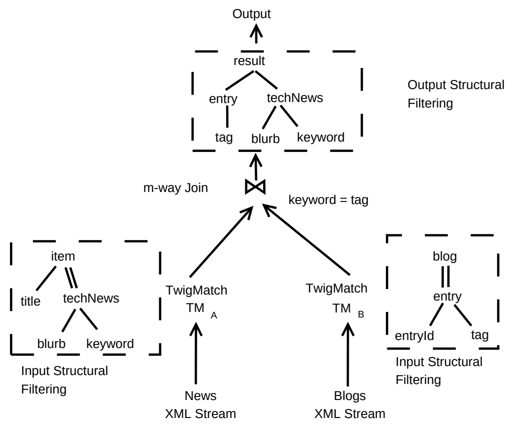


Figure 7.1: Query Execution Plan

7.1.1 Twig'n Join Algortihm

Algorithm 2 Twig'n Join Algortihm

```
1: for (i=0; i < n; i++) do
2:     TwigMachine $TM_i$ = CreateTwigM_Machine($T_i$,$S_i$)
3:     HashPartition $Ht_i$ = CreateHashPartitions()
4: end for

5: while ( XML fragments are available ) do
6:     xmlfrag = Select(S)
7:     MultiWayJoin(xmlfrag)
8:     $Ht_{src}$.insert(xmlfrag)
9: end while
10: return (R, Results)
```

Algorithm 2 shows the details. In Line 1 to 3, we create a TwigM machine and a hash partition for each of the XML data streams. . The TwigM machines exposes an iterator-style (i.e. *getNext()*) interface in order for the TnJ algorithm to continuously get the next XML fragments that have been matched using T_i ($0 \leq$ i < n, where n denotes the number of XML data streams). In Line 6, the *Select()* checks the availability of XML fragments from the various TwigM machines. In Line 7, the XML fragment is used in a multi-way join on the remaining *n - 1* hash partitions. Whenever memory is full, some of the XML fragments in the hash partitions will be flushed to disk. This is checked prior to the insertion of the XML fragment into the corresponding hash partitions (Line 8). Algorithm 4 describes how the insertion and flushing is done. Then, the XML fragment is inserted into its own partition. We make use of the Berkeley-DB [OBS99] hash function for computing the hash value for each XML fragment.

7.1.2 Twig Matching

In this section, we discuss the basic structure of TwigM machine [CDZ06].

Given the twig query Q, a TwigM machine, M, is created (Figure 7.3(b)). M consists of machine nodes n_i. For each node n_i, there is an edge e_i which connects it to its parent node ($1 \leq \mathrm{i} \leq |Q|$, where $|Q|$ refers to the number of tags specified in the query). Depending on whether it is a parent-child or ancestor-descendant relationship specified in the query, the edge is annotated with 1 (parent-child) or $\geq$1 (ancestor-descendant). For example, in Figure 7.3(b), we can see that the edges are all annotated with $\geq$1. This corresponds to the query Q. In addition, a stack is associated with each of the machine nodes. For a non-leaf machine node, an entry of the stack is a triple <N, C, B>, where N refers to a XML tag that is inserted, C is a candidate solution list, and B is a boolean array. For a leaf machine node, an entry of the stack only consists of just < N>. For a node with b children, B consists of b boolean variables. Initially, the b boolean variables are all initialized to be false. M is then used to process the twig queries and deliver the results (in the form of XML fragments) whenever a match occurs.

Whenever portions of an XML documents satisfy the structural constraint expressed by the twig query, the TwigM machine outputs the results as XML fragments. The results are output whenever the structural constraints are met. Hence, the XML fragments can be delivered progressively for join processing.

To illustrate how the TwigM machine works, consider the twig query, Q, (Figure 7.3(a)) and the following XML document D, which is streamed in document order from a remote data source. In the XML document D, the unique identifiers for the various tags (given in parenthesis) are Section (S1 and S2), Title (T1 and T2) and

Figure (F1, F2).

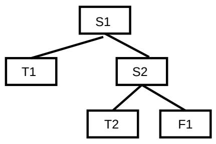
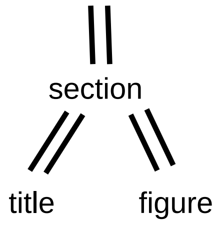
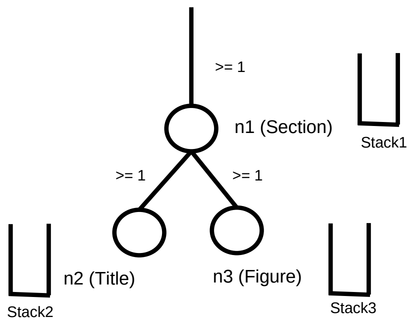


Figure 7.2: XML Document, D



(a) twig Query, Q



(b) TwigM Machine, M

Figure 7.3: twig Query and TwigM Machine Example

Whenever the start element for a tag is encountered, we will first push it into its corresponding machine node stack. It is important to note that an entry is pushed onto its corresponding tag if the level differences fulfill the edge condition (i.e. whether the parent-child or ancestor-descendant condition is met). For leaf machine nodes, whenever the end element of a tag is encountered, we will pop it from its stack, and add it to the candidate list C for all entries in its parent machine node stack. Afterwhich, we set the corresponding entry in the boolean array B of the parent machine node to be True. For non-leaf machine nodes, we will only pop an entry from the stack only when the entire boolean array B is True. For more details on the TwigM algorithm, refer to [CDZ06].

Let us illustrate how TwigM works using the XML document D given earlier. For document D, tag <S1> arrives. We push <S1, C={}, FF> into Stack1. Next, <T1> arrives. We first check the level difference for T1 and the nodes in Stack1. If the level difference fulfills the ancestor-descendant edge condition imposed, we will push <T1> onto Stack2. Figure 7.4(a) shows a snapshot of the three stacks.

Next, we encountered the end element for </T1>. Since node n2 is a leaf machine node, we pop the stack entry for <T1> (and its associated content) from the stack, and add it to the candidate list of all entries in Stack2 where the condition of for query Q is fulfilled. We set the B array for each of the entry to be T, denoting the left branch of the machine node n1 (Section) has been matched. Figure 7.4(b) shows the content of the various stack at this point.

The start element for <S2> arrives. We add it to Stack1. We then encounter the start element for <T2>. We check all entries in Stack1 to see whether any of the entries and <T2> fulfills the edge condition. We add <T2> to Stack2 if the edge condition is fulfilled. Figure 7.4(c) shows the content of the stack. Next, the end element for </T2> arrives. We pop the entries from Stack2 and add it to the entries in Stack1. Figure 7.4(d) shows the content of the stacks. The start element <F1> arrives. After checking that it fulfills the edge condition, we add it to Stack3 (Figure 7.4(e)). Afterwhich, the end element </F1> arrives. We pop the entry from Stack3 and add it to the candidate list of its parent. We also set the boolean value to T for the right branch (Figure 7.4(f)). Notice that the B boolean array of the entries in Stack1 are all set to TT.

Finally, the end element </S1> arrives. Since the boolean array B for the entries are TT, we therefore output the solution for the twig query as XML fragments <S1,T1,F1>, <S1,T2,F1>, and <S2,T2,F1>.

7.1.3 Join Processing

Results from the twig matching on the multiple XML streams are fed to the hash-based progressive join. We make use of the generic Result Rate-based flushing (RRPJ) technique used in [TBL07c]. During join processing, RRPJ is used to determine the XML fragments to be flushed to disk whenever memory is full. An important characteristic of RRPJ is that by using statistics based on the result output statistics, it can be generalized gracefully for many data models (as shown in [TBL06](spatial data) and [TBL07b](high-dimensional data)).

Probing

Algorithm 3 shows how a newly arrived XML f_d is used to probe the corresponding hash partition from the other data stream (Line 1). Based on the join predicates defined, we check each of the XML fragments found in the partition (Line 3-6). Results are output whenever the join predicates are satisfied. In addition, a counter, *numResults*, keeps track of the results produced by each of the partitions. The counter is updated when all the results have been produced (Line 7).

Algorithm 3 Probing

```
1: $p$ = findPartition($f_d$)
2: numResults = 0
3: for ( XMLFragment $f$ in $p$) do
4:     if ($f$ and $f_d$ satisfies the join predicate) then
5:         $R = R \cup (f_d,f)$
6:         numResults++
7:     end if
8: end for
9: Update statistics for $p$
10: return ()
```

Insertion and Flushing

Algorithm 4 shows the insertion algorithm. The hash value for an XML fragment is computed. The XML fragment is then inserted into its corresponding hash partition (Line 1). Whenever memory is full, the *FlushDataToDisk*() routine flushes some of the in-memory XML fragments to disk. The number of XML fragments to be flushed is determined by a user-defined parameter, *NumFlush*.

In order to determine which partitions to be flushed, each of the ith hash partitions ($1 \leq$ i $\leq$ n, where n is the total number of partitions), maintains a counter measuring its potential to produce results. This determines the partitions to be flushed.

We first present a naive extension of RPJ for determining the XML fragments to be flushed to disk whenever memory is full. In the naive extension to RPJ for XML

Algorithm 4 Insertion and Flushing

```
1: $p$ = findPartition($f_d$)
2: if ( memoryIsFull()) then
3:     FlushDataToDisk()
4: end if
5: Insert $f_d$ into $p$
6: return ()
```

(called Twig-RPJ), we keep track of the the RPJ value - number of XML fragments in a partition divided by the total number of XML fragments that have arrived. The partner partition (that is the matching partition in the other streams) to the partition with the smallest values is flushed.

We also make use of the Result-rate based Join (RRPJ) flushing technique described in [TBL07c]. When making use of the Result Rate-based Flushing (RRPJ), we keep track of the Th_i value. In RRPJ, the Th_i value is an estimate of the productivity of the i-th partition (with 1 ≤ i ≤ n, where n is the total number of partitions used to store the XML fragments).

$$Th_i = \frac{Ri}{N_i} \tag{7.1}$$

where R_i and N_i denotes the total number of results produced, and the total number of XML fragments for the i-th partition respectively. Twig'n Join flushes the partitions with the smallest Th_i values, until a user-defined number of tuples to be flushed is reached.

Multi-way Join

In this section, we discuss how we can generalize Twig'n Join for processing XQuery queries on multiple XML streams. Each XML fragment, produced by the TwigM machine, consists of the XML data and a bitmap (i.e DoneBitmap) that is used to determine whether the XML fragment has been used to probe the other partitions. *DoneBitmap* consists of n bits. Figure 7.5 shows the structure of the XML fragment. When the XML fragment first arrives, the bit corresponding to each own partition is set to 1. Whenever the XML fragment is used to probe the hash partitions for the other XML streams, the bit corresponding to the hash partition is set to 1 when it can be used to join with at least one other XML fragment in the partition. It is set to 0 otherwise. When all the bits of the *DoneBitmap* are set, the XML fragment is output as a result. In this thesis, we consider only the case where the join predicate is the same for all the XML streams.

Existing multi-way join techniques for relational equi-join, such as MJoin [VNB03], can be used as to handle the multi-way between the XML fragments that are produced. The performance of the multi-way join is dependent on the probing sequence. For example, MJoin sorts the hash partitions based on their respective join selectivity. The key intuition is that by probing partitions with a low join selectivity first, it filters away tuple that will not generate any result early. This helps to reduce the number of unnecessary probes to the remaining un-probed hash partitions. However, it is difficult to determine the join selectivities if the inputs to the multi-way join consists of intermediate results from a pipelined process. For example, the XML fragments are produced by the TwigM machines. Even if the join selectivity of the join attribute for the base XML streams can be accurately determined, it is not straightforward to determine the join selectivity of the intermediate XML fragments produced. In addition, determining the join selectivity apriori might not be useful if the join selectivity changes during the lifetime of the multi-way join.

In order to deal with the problem of determining an effective probing sequence for the multi-way join, we propose a novel technique, called Result-Oriented Probing (RoP). RoP dynamically determines the order of the hash partitions to be probed in the multi-way join. RoP tracks the number of partial results that are produced by each hash partition. Whenever a XML fragment f is used to probe a hash partition, a partial result is generated if the bits of the *DoneBitmap* for f have not been completely set to 1. In contrast, a complete result is generated if all the bits of the *DoneBitmap* for f are set to 1.

Algorithm 5 MultiWayJoin with RoP

```
$ProbeSequence$ = SortHashPartitionsAsc()
for (i=0; i < n; i++) do
  idx = $ProbeSequence_i$
  numResults = $Ht_{idx}$.probe($f_d$)
  if ( numResults == 0 ) then
    break;
  end if
end for
```

7.2 Performance Evaluation

We implemented all the algorithms in C++, and conduct the experiments on a Pentium 4 2.4 Ghz PC (1GB RAM). Similar to [CDZ06], we make use of the SAX Parser - Expat [Cla03]. Unless otherwise stated, the parameters presented in Table 7.1 are used for the experiments. We compare the performance of Twig-RPJ, Twig'n Join (TnJ). In addition, we also included a Random method as a baseline. Whenever memory is full, the Random method randomly selects a partition (containing XML Fragments) to be flushed to disk.

Parameter	Values
Disk Page Size	40960 bytes
Memory Size, M	10% of data size
Number of entries per page	31
Number of XML Fragments flushed to disk	10% of M
Number of hash partitions	1024

Table 7.1: Experiment Parameters and Values

7.2.1 X007

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using synthetic datasets generated using X007 [BDL$^+$01]. We set the X007 parameters as given in Table 7.2. We varied the X007 parameter NumConnPerAtomic for values 3, 6 and 9. For each NumConnPerAtomic value, we generated two datasets to simulate

X007 Parameters	Values
NumAtomicPerComp	20
NumConnPerAtomic	3,6,9
DocumentSize(bytes)	500
ManualSize(bytes)	2000
NumCompPerModule	50
NumAssmPerAssm	3
NumAssmLevels	5
NumConnPerAssm	3
NumModules	1

Table 7.2: X007 Parameters

the two XML streams. In this experiment, we join the type IDs reference for the Connections. The twig query given below is used for both datasets.

- //Connection[type][AtomicPart]

As the graph for values 3,6 and 9 exhibits similar trends, we present the results for NumConnPerAtomic = 9 in Figure 7.7. From the figure, we can observe that all the methods (TnJ, Twig-RRPJ and Random) perform similarly. This is the case because the values of the join attribute for the XML fragments are uniformly distributed. Thus, regardless of the XML fragments that are flushed to disk, there is no impact on the overall throughput. Similarly, the same observations are made in [TBL07c] for uniformly distributed relational data.

7.2.2 XMark

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using synthetic datasets generated using XMark [SWK^{+}02]. Table 7.3 shows the size of the XMark datasets, and also the number of XML fragments extracted by the TwigM machine on-the-fly. The fragments are then used in the join of XML fragments.

XMark generates a single XML document consisting of information on the annotation, person, category, closed auction, open auction and the items. For the purpose of the experiments, we extracted out the details of the items and closed auctions

XMark Factor, λ	Items Fragments	ClosedAuctions Fragments	Total Fragments	Dataset Size(MB)
0.2	4350	1950	6300	20
0.4	8700	3900	12600	38
0.6	13044	5845	18889	57
0.8	17400	7800	25200	76
1.0	21750	9750	31500	94
2.0	43500	19500	63000	187

Table 7.3: XMark Dataset Information

into 2 separate XML files. This is used to simulate two XML data streams. In this experiment, we join the item IDs reference of the closed auctions with the items. The join attribute is *id* (string). The following twig queries are defined on the Item and Closed Auctions streams respectively.

- **Items**: //item[id][name]
- **ClosedAuctions**: //closed_auction[itemref/id][price]

In these experiments, we varied the scaling factor of XMark, λ, between 0.2 and 2.0. We present the results for varying λ in Figure 7.6. In all cases, Twig'n Join outperforms Twig-RPJ and the random flushing strategy by a large margin.

7.2.3 TPCH

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using XML datasets which were converted from datasets generated by TPC-H [XML02]. We join the data between Orders and Customer. We specify CUST_KEY as the join attribute. The characteristics of the dataset is tabulated in Table 7.4. The following twig queries are defined on the Orders and Customer XML data streams.

- **Orders**: //T[CUSTKEY][O_ORDERSTATUS]
- **Customer**: //T[CUSTKEY][C_NAME]

Dataset	Number of Elements	DataSet Size
orders.xml	150001	5MB
customer.xml	13501	503KB

Table 7.4: TPC-H Benchmark (XML version)

From Figure 7.8, we can observe that Twig'n Join outperforms Twig-RPJ significantly. This further shows that Twig'n Join is able to keep XML fragments that have a higher probability to produce results in-memory. Thus, this enables it to be able to produce more results compared with Twig-RPJ.

7.2.4 DBLP vs SIGMOD Record

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using two real-life datasets. We used the DBLP dataset (scaled down to 29MB), and SIGMOD Record (467K) [XML02]. In the experiments, we join on the author attribute (i.e. we want to find authors who have published in SIGMOD Record and have at least one publication listed in DBLP). The following twig queries are defined on the SigmodRecord and DBLP data streams.

- **SigmodRecord**: //issue[volume][//article[title][//author]]
- **DBLP**: //inproceedings[author][title]

From Figure 7.9, we can see that Twig'n Join outperforms the Twig-RPJ method when approximately 24% of the XML fragments have arrived. We can also observe that the number of result fragments produced increases quickly between 16% - 24% of the XML fragments have arrived. This is because the TwigM machine has not produced sufficient XML fragments which can be joined between the two XML data streams. Beyond 24%, there are sufficient XML fragments available from the DBLP XML data streams in-memory to be joined. Thus, the number of results produced grows linearly beyond that.

μ	Number of fragments	Size(MB)
0.2	11420	2.7
0.4	22612	5.3
0.6	34180	8.0
0.8	45377	11

Table 7.5: Sizes of BioExpts

7.2.5 Swiss-Prot

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using a real-life dataset (Swiss-Prot, available at [XML02]) and a synthetic dataset (BioExpts). Using the Swiss-Prot dataset, we generate the BioExpts dataset to simulate the details of biological experiments conducted using the protein sequence found in Swiss-Prot. The BioExpts XML file consists of the following information: (1) Experiment ID (ID), (2) Researcher Userid (Researcher), (3) Accession Number (AC) and (4) Observation. The researcher userid and observation consists of randomly generated strings of length 10 and 100 respectively. Figure 7.10 shows a snippet of the XML generated.

When generating the synthetic dataset, the parameter μ, controls the probability in which an Accession Number from the Swiss-Prot dataset is used in an experiment. When $\mu = 0.0$, then none of the Accession Number are used in the experiments (i.e. no results produced during the join of the Swiss-Prot and the BioExpts dataset). When $\mu = 1.0$, all Accession Numbers are used in the experiments. In other words, μ controls the join selectivity. We vary μ from 0.2 to 0.8. In the experiments, we join on the Accession (AC) Number attribute. The size of Swiss-Prot is 110MB, and the sizes of the synthetic datasets for varying μ are presented in Table 7.5.

The following twig queries are defined on the SwissProt and synthetic dataset.

- **SwissProt**: //Entry[AC][Species]
- **BioExpts**: //Expt[/Info/ID][AC]

From Figure 7.11, we can observe that Twig'n Join consistently outperforms Twig-RPJ for varying μ. Another interesting observation is that when $\mu = 0.8$, the baseline Random method performs better than Twig-RPJ. This shows that the naive Twig-RPJ is not as effective in determining the XML fragments to be flushed to disk whenever memory is full.

7.2.6 Multi-way XML Join

In this section, we compare the performance of the multi-way join using various probing techniques. These includes: (1) RoP (2) Sequential and Apriori. RoP uses the dynamic probing sequence described in Section 7.1.3. In the Sequential probing strategy, we probe the hash partitions in the order in which the XML streams arrive. In the Apriori strategy, we assume that we know the join selectivity of each of the XML streams. We then probe the hash partitions in order of increasing join selectivity. Thus, hash partitions with lower join selectivity are probed first. We evaluate the performance based on two metrics. Firstly, we count the total number of probes on the hash partitions. Secondly, we measured the time taken to produce results.

The XML streams used in this experiment is generated as follows. We first extracted all the name of authors from SIGMOD Record. Using the names of authors, we generated a reference XML stream in which consists of blog entries written by the authors. Next, we generated the other XML streams to be used in the multi-way join by controlling the selectivity, μ. μ determines the probability that a author from the reference XML stream is included in the stream to be generated. We vary μ between 0.0 to 1.0. When $\mu = 0.0$, none of the authors from the reference XML stream are included. When $\mu = 1.0$, all the authors from the reference XML streams are included. Various m-way joins are evaluated (m varies between 3 to 5).

From Figure 7.12(a), we can observe that dynamic result-oriented probing (RoP) outperforms the Sequential probing technique. In addition, RoP performs almost as well the Apriori strategy. This shows that the dynamic RoP technique is effective even without prior information on the join selectivities. From Figure 7.12(b)-(d), we can observe that RoP outperforms the Sequential probing technique. This commensurates with the findings from Figure 7.12(a). As a result of the significant reduction on the number of unnecessary probes, RoP takes less time to produce the same number of results.

7.3 Summary

In this chapter, we propose a practical approach for progressive processing of (FWR) XQuery queries on multiple XML streams, called Twig'n Join. We decompose a (FWR) XQuery query into a query plan consisting of twig queries and join processing. The twig queries are used for processing the structural constraints. The hash-based join operator is used to process the join predicate constraints. The novelty of this approach compared, to conventional XQuery processing, lies in the decomposition of the XQuery queries into several independent components. This reduces the complexity for the design of XQuery query processing algorithms. Though we show this for XQuery queries involving joins, the technique can be applied to the the various type of (FWR) XQuery queries as well.

Due to the large amount of streaming XML data, it is infeasible to keep all the XML data in-memory during join processing. We make use of the RRPJ method [TBL07c] to flush the XML data whenever memory is full. In addition, we introduce a novel dynamic probing technique, called Result-Oriented Probing (RoP), which determines an optimal probing sequence for the multi-way join. This significantly reduces the amount of redundant probing for results. Experiment results show that Twig'n Join is indeed effective and efficient for the processing of both synthetic and real-life datasets.

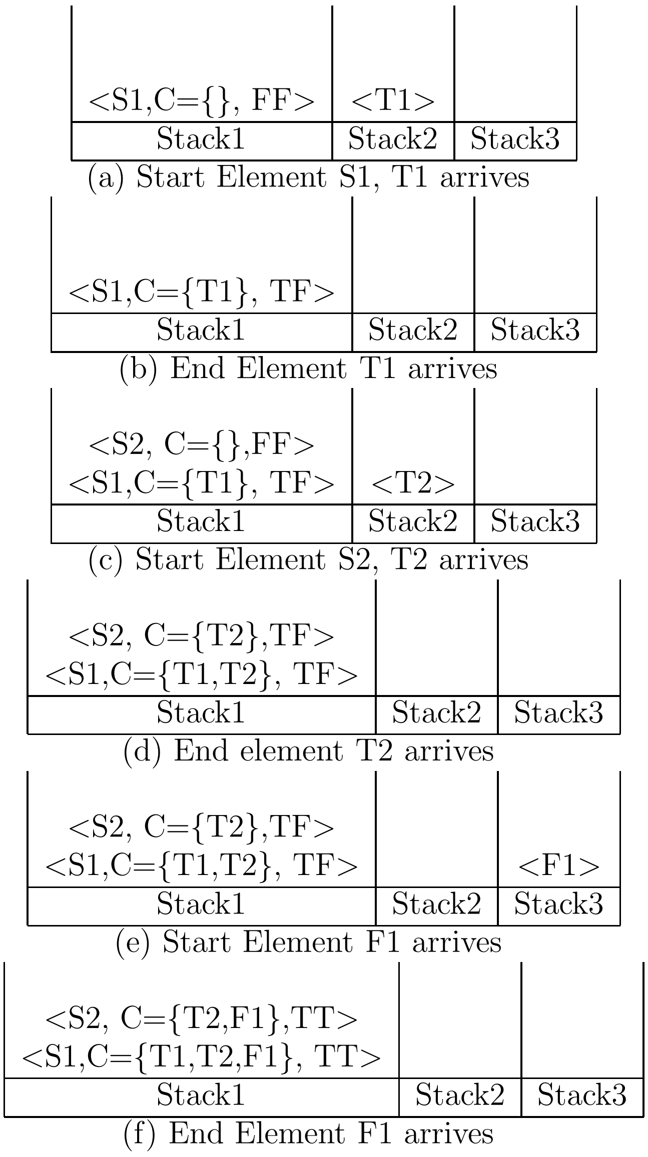


Figure 7.4: Snapshot of the stack

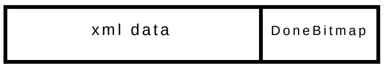
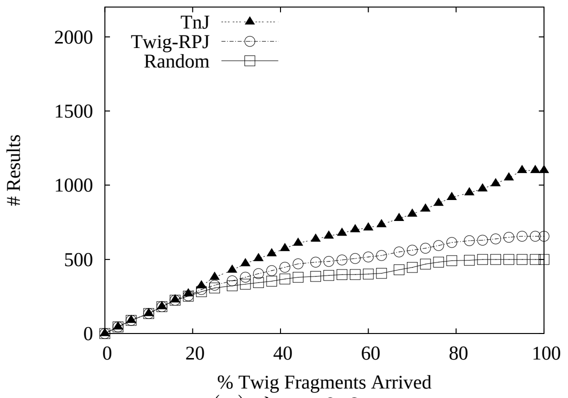
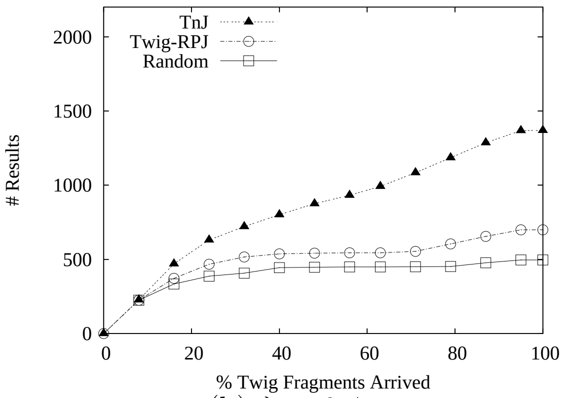


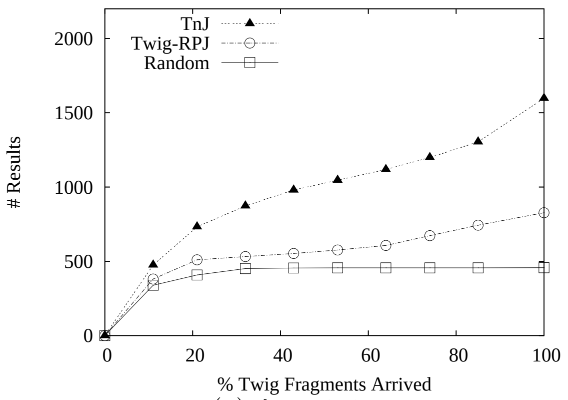
Figure 7.5: XML Fragment Structure



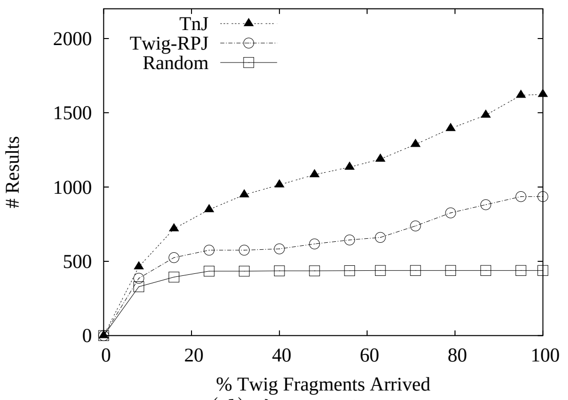
(a) $\lambda = 0.2$



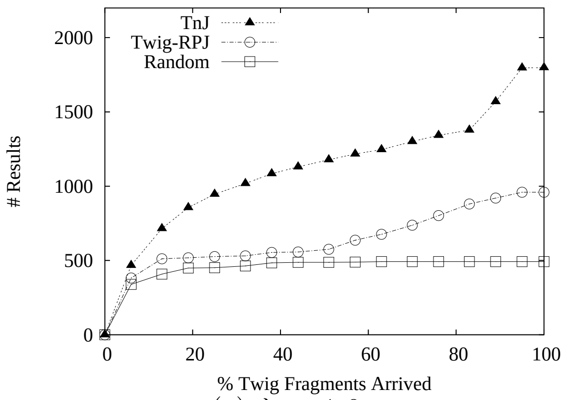
(b) $\lambda = 0.4$



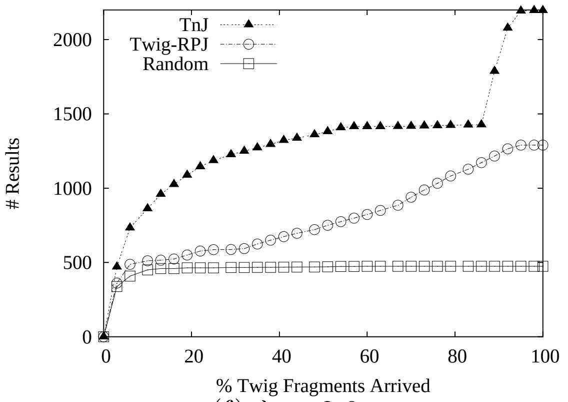
(c) $\lambda = 0.6$



(d) $\lambda = 0.8$



(e) $\lambda = 1.0$



(f) $\lambda = 2.0$

Figure 7.6: Varying XMark Factor, λ

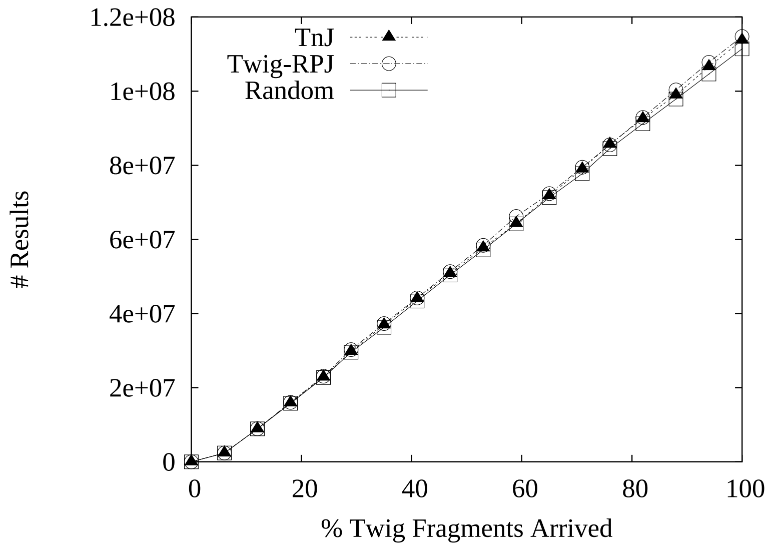


Figure 7.7: X007

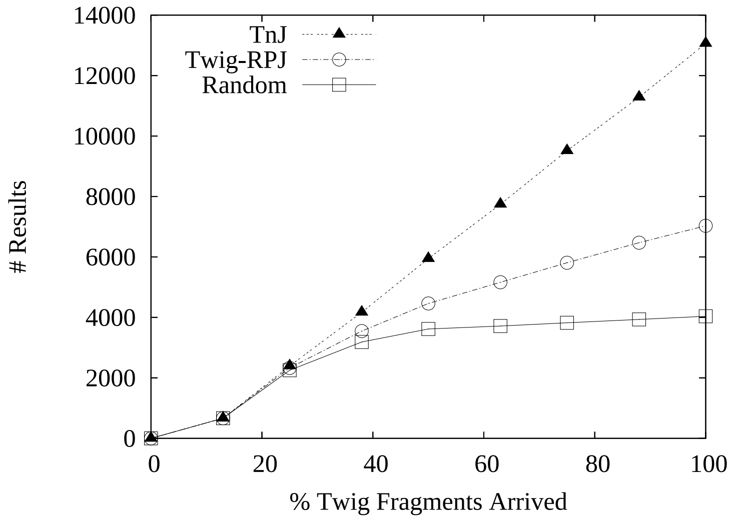


Figure 7.8: TPCH (XML Format)

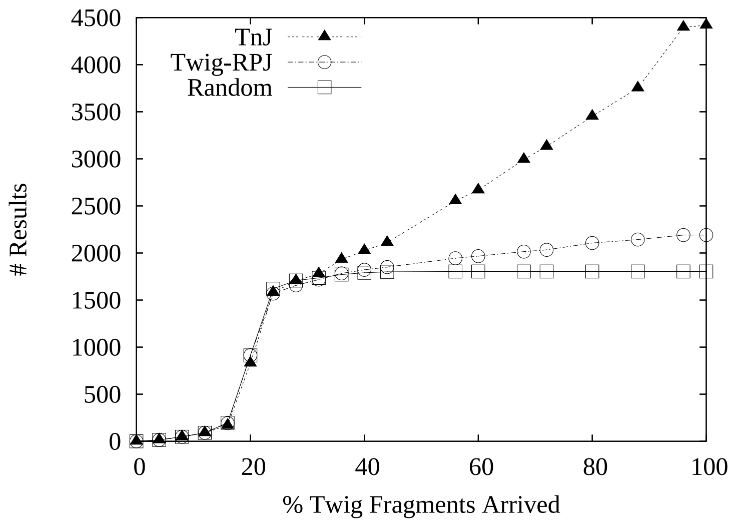
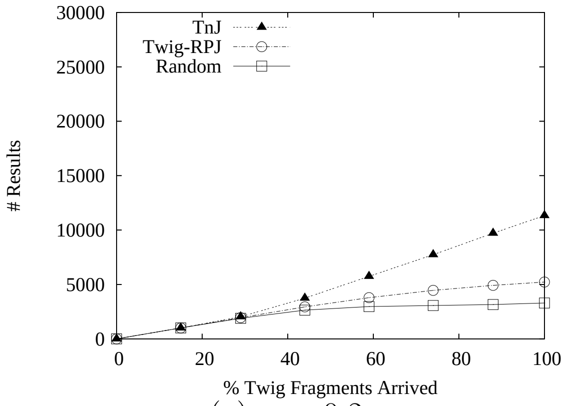


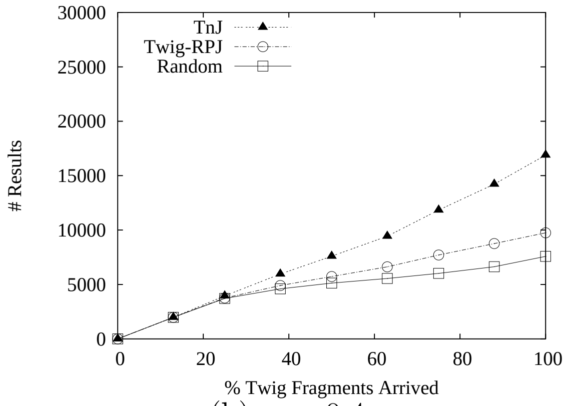
Figure 7.9: DBLP vs SIGMOD Record

```
<BioExpts>
<Expt>
  <Info>
     <ID>1</ID>
     <Researcher>gXKhK4hkXP</Researcher>
  </Info>
  <AC>P14914</AC>
  <Observation>ABzAW71t ...</Observation>
</Expt>
<Expt>
  <Info>
     <ID>2</ID>
     <Researcher>sHqj5LraCT</Researcher>
  </Info>
  <AC>Q26540</AC>
  <Observation>CjyIl0yjp6Q ...</Observation>
</Expt>
...
</BioExpts>
```

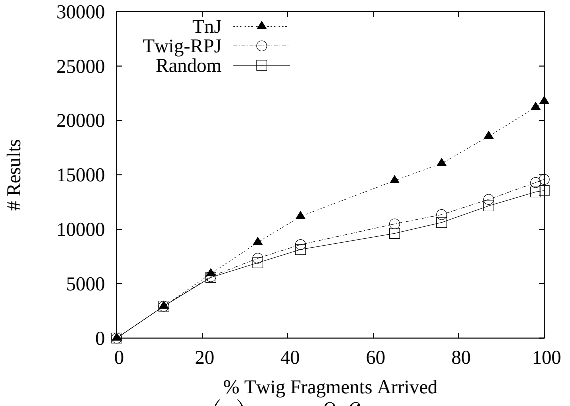
Figure 7.10: Synthetic Dataset based on Swiss-Prot



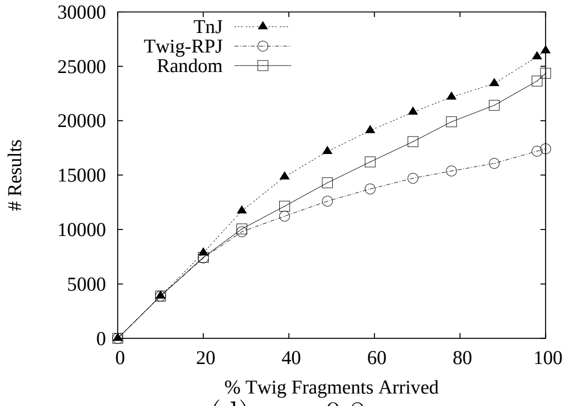
(a) $\mu = 0.2$



(b) $\mu = 0.4$

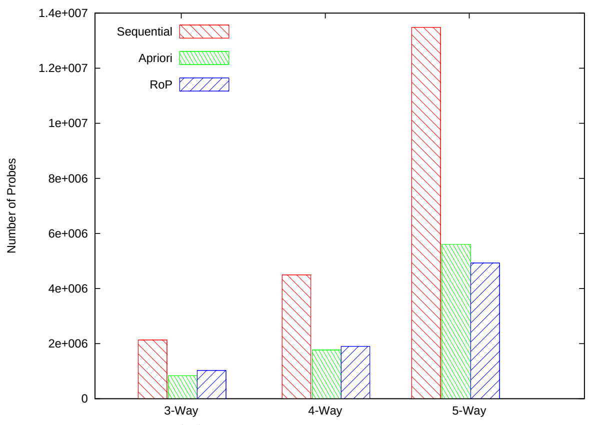


(c) $\mu = 0.6$

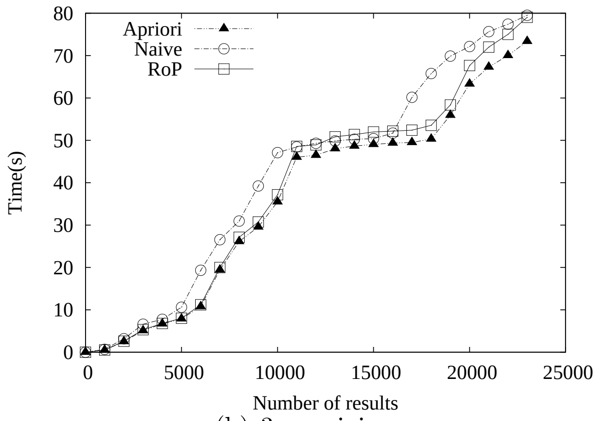


(d) $\mu = 0.8$

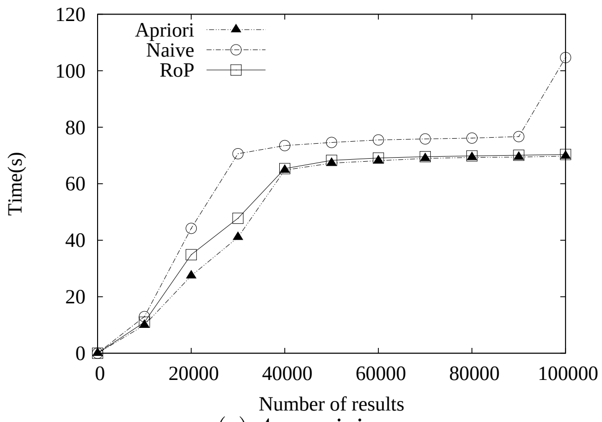
Figure 7.11: Swiss-Prot vs BioExpts : Varying μ



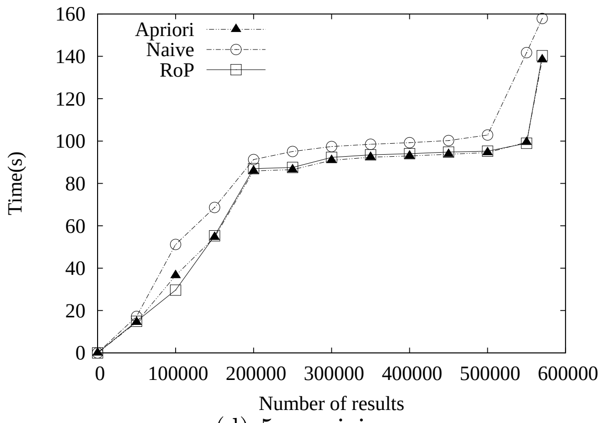
(a) Number of Probes



(b) 3-way join



(c) 4-way join



(d) 5-way join

Figure 7.12: Multi-Way Join (with different probing sequence)

Chapter 8

Progressive Approximate Joins

Users often do not require a complete answer to their query but rather only a sample. They expect the sample to be either the largest possible or the most representative (or both) given the resources available. We call the query processing techniques that deliver such results 'approximate'. Processing of queries to streams of data is said to be 'progressive' when it can continuously produce results as data arrives. In this thesis, we are interested in the progressive and approximate processing of queries to data streams when processing is limited to main memory. In particular, we study one of the main building blocks of such processing: the progressive approximate join. We devise and present several novel progressive approximate join algorithms. We empirically evaluate the performance of our algorithms and compare them with algorithms based on existing techniques. In particular we study the trade-off between maximization of throughput and maximization of representativeness of the sample.

8.1 Introduction

In many data stream applications [BW01, BBD$^+$02, CcC$^+$02], users are concerned with rapid production of results that they are ready to give up completeness of the result. In this case, users may prefer results that can be produced in main memory only. In other words, users often do not require a complete answer to their query but rather only a sample. They expect the sample to be either the largest possible (i.e.

quantity), the most representative (i.e. quality) or both? They may need to seek a compromise between quality and quantity, given the resources (main memory) available. We call the query processing techniques that deliver such results 'approximate'.

Join algorithms being the keystones of query processing, we are interested here in progressive approximate join algorithms. The reference progressive approximate join is *Prob* introduced in [DGR03] and its extended version [DGR05]. The authors introduce the notion of maximum subset (MAX-Subset) which leads to similar strategies as the ones used by progressive algorithms such as RPJ [TYP+05] and RRPJ [TBL07c] to maximize the size of the set of results produced, quantity. We show that the performance of *Prob* can be improved by stratifying the memory available. We propose *ProbHash*, a direct extension of *Prob*, in which the memory is hash partitioned and an approximate version of our progressive algorithm RRPJ also using hash partitioning. Interestingly, the authors of [DGR03] have disqualified reservoir sampling based methods based on the extreme scenario given in [CMN99] without further experiments. We show that this disqualification is mistaken. We propose a reservoir sampling-based approximate progressive join, that we call Reservoir Approximate Join (RAJ), and its stratified version *RAJHash*. We show that these algorithms favor the representativeness of the set of results produced and ensure better quality than the other algorithms.

The rest of the chapter is organized as follows. We discuss the notions of quantity and quality of results produced in Section 8.2. We present the proposed algorithms in Section 8.3. In Section 8.4, we empirically evaluate the performance of our four algorithms and compare them with *Prob*. In particular we highlight and discuss the trade-off between quantity and quality. We conclude and discuss future work in Section 8.5.

8.2 Measuring Performance

8.2.1 What do We Measure?

There are two ways to measures the performance of an approximate algorithm. If we are interested in quantity, the measure of performance for the algorithm is the amount of results produced. If we are interested in quality, we need to measure the similarity between the data distribution of the complete set of results and the data distribution of the set of results produced. Because we are interested in progressive algorithms, performance is not a unique number but a function of time. It is measured in term of throughput, quantity over time, when size matters. It is measured in terms of quality over time (quality throughput), when quality matters. If both quantity and quality matter, we need both functions. Notice that the comparison of both functions by looking at quantity as a function of quality (or vice versa) at given points in time visualizes the compromise realized by a given algorithm.

We considered defining a combined measure of quantity and quality (similarly to the F-measure, which combines recall and precision). Unfortunately, our measure of quality using JS Divergence or any comparable statistical measures is unbounded, and cannot be normalized.

8.2.2 How do We Measure Quality?

In order to measure quality, we need to compare two data distributions. We can compute, combine and compare any statistics and obtain more or less significant measurements at different level of granularity.

A reasonable metric is the Mean-Square Error (MSE) between the normalized histograms of the complete results and result produced by the approximate join. Another metric for comparing data distribution is the Jensen-Shannon divergence [Lin91]. The Jensen-Shannon divergence (JSD) determines the similarity (or divergence) between two probability distributions. In this chapter, we make use of the JSD measure.

MSE

We first discuss the MSE measure. The MSE measure measures the error differences between the actual and observed results produced. In the approximate join scenario, the actual results refer to the results produced if the entire join is computed (or when the memory is unlimited and all data fit into main memory). The observed results refer to the results produced by the approximate join method. In order to ensure a fair comparison between the actual and observed result distribution, we compare the normalized frequency instead of the actual frequency for each join attribute value. Let the total number of results produced by the complete and approximate join be $|R|$ and $|R'|$ respectively. For each value $v_i \in V$, where V denotes the domain of the join attribute, and $1 \leq i \leq |V|$. $|v_i|$ and $|v_i^{'}|$ denotes the number of actual and observed results with value v_i. For each join attribute v_i, the normalized value for the complete and approximate joins is given by $\frac{|v_i|}{|R|}$ and $\frac{|v_i^{'}|}{|R'|}$ respectively. The MSE between the complete join J and approximate join J' is given by

$$MSE(J, J') = \sum_{i=1}^{V} (\frac{|vi|}{|R|} - \frac{|v_i^{'}|}{|R'|})^2 \tag{8.1}$$

Jensen-Shannon Divergence

In probability and information theory, the Kullback Leibler (KL) and Jensen-Shannon divergence are used to measure the similarity between two probability distributions, P and Q. We use the Jensen-Shannon divergence to measure the similarity between the actual (P) and observed result (Q) distribution. We measure the result quality produced by the approximate join using the Jensen-Shannon divergence. The Jensen-Shannon divergence measures the similarity between the actual result distribution (produced by a join where all tuples fit in memory) and the approximate join result distribution. Let p(v_i) = $\frac{|v_i|}{|R|}$ and q(v_i) = $\frac{|v_i^{'}|}{|R'|}$. Before defining the Jensen-Shannon divergence, we first define the KL divergence, given as follows:

$$D_{KL}(P||Q) = \sum_{i=1}^{V} p(v_i) \log(p(v_i)/q(v_i)) \tag{8.2}$$

The Jensen-Shannon divergence is given by

$$D_{JS}(P||Q) = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M) \tag{8.3}$$

where M $= \frac{1}{2}(P + Q)$

The goal is to minimize either the MSE or the JS divergence. When the value for either MSE or JS divergence is zero, the result distributions from the complete and approximate joins are exactly the same.

Given two approximate join methods, J_1 and J_2, we say that J_1 produces better quality results than J_2 if the *QMeasure*(J_1) < *QMeasure*(J_2). *QMeasure*(Z) refers to either computing MSE(Z) or D_{JS}(Z). Z refers to an arbitrary approximate join method.

8.3 Solution

In this section, we describe five methods for performing approximate joins: (1) *Approximate RRPJ* (ARRPJ), (2) *Prob*, (3) *ProbHash*, (4) *Reservoir Approximate Join* (RAJ) and (5) *Stratified Reservoir Approximate Join* (RAJHash).

We first present the key idea for an existing progressive approximate join algorithm, *Prob*. Next, we propose the modification of an existing progressive join algorithm for approximate join processing, called *Approx-RRPJ*. Lastly, we propose three new algorithms (*ProbHash*, *RAJ* and *RAJHash*). *ProbHash* aims to maximize the result quantity as well as improve the overall throughput. Both *RAJ* and *RAJHash* are designed to optimize the result quality.

8.3.1 Approximate Join Framework

We first discuss a general framework for designing approximate join algorithms which explore the tradeoffs between result quantity and quality.

Given two streams S_1(A,B) and S_2(B,C), where A, B and C are attributes of the data streams. Let the *i-th* tuple from S_1 and the *j-th* tuple from S_2 be denoted by $t_{S1}(a_i,b_i)$ and $t_{S2}(b_j,c_j)$ respectively. An approximate join is used to join the tuples from the two streams. The size of the memory available for query processing is small relative to the size of the data streams, which can be unbounded. When a new tuple arrives and memory is full, we will need to selectively discard some tuple(s) from memory. An important design criteria for an effective approximate join algorithm is to determine how tuples are discarded.

We first consider approximate join algorithms which maximizes the quantity of the results produced. We call such a algorithm DP_X(k), which discards k tuples whenever memory is full. The goal of the DP_X(k) policy is to maximize the expected size of the result subset. To achieve this, we can model the probability of the join attribute value(s) for tuples arriving on both streams. Let the arrival probabilities be P_{S1}(B) and P_{S2}(B) for streams S_1 and S_2 respectively. Whenever a tuple arrives, we assign a priority to the tuple based on the arrival probabilities from the corresponding stream. For example, when a tuple $t_{S1}(a_i,b_i)$ arrives, its priority value is given by $P_{S2}(b_i)$. Similarly, the priority of a tuple $t_{S2}(b_j,\ c_j)$ can be computed using $P_{S2}(b_j)$. A possible implementation for DP_X(k) is to maintain two priority queue (in ascending priority order) for the data streams. Whenever memory is full, DP_X(k) discards the first k tuples taken from both streams. The intuition is that by keeping in memory tuples which have higher probability of joining with tuples from the other stream, the expected number of results produced will be maximized [TYP$^+$05].

Next, we consider approximate join algorithms which are sampling-based. The goal is to optimize the quality of the results produced. We call such a algorithm DP_Y. DP_Y continuously maintains a random uniform sample for each of the data streams. When the memory is not full, tuples are inserted into the respective reservoirs. When memory is full, DP_Y determines whether the newly arrived tuple should be discarded, or be used to replace a tuple from the reservoir. Suppose the size of the memory is M, which is divided equally between the two streams S_1 and S_2. Suppose n_{S_1} and n_{S_2} tuples have arrived for stream S_1 and S_2 respectively. We assume that the number of tuples that have arrived for each stream is greater than the available allocated memory (i.e. n_{S_1} > (M/2), and n_{S_2} > (M/2)). A newly arrived tuple $t_{S1}(a_i,b_i)$ has a $\frac{(M/2)}{n_{S_1}}$ chance of being used to replace a tuple in the reservoir. Similarly, for a tuple from S_2. Even though DP_Y might not maximize the number of results produced, the quality of the results produced could be much better than DP_X(k). This is because DP_Y ensures that the uniformity of the samples for each of the data streams. When a new tuple arrives, it is used to probe the corresponding reservoir. Mindful readers might note that DP_Y might not work well for skewed data streams if the memory is allocated equally between the two reservoirs. In this thesis, we show how we can tackle this problem by dynamically allocating memory for the reservoirs.

8.3.2 Approximate RRPJ (ARRPJ)

The Result-Rate Based Progressive Join (RRPJ) [TBL07c] was proposed as a progressive join algorithm. It builds statistics on the result distribution of the hash partitions. The goal of RRPJ is to maximize the number of results produced by using the result distribution statistics to determine the non-productive tuples to be flushed to disk whenever memory is full. In RRPJ, when all the tuples have arrived, a cleanup phase is invoked to compute the complete results for the join query.

In order to build a progressive approximate join, we modify RRPJ so that it consists of the in-memory processing phase. We call this join algorithm, *Approximate RRPJ (ARRPJ)*. Whenever memory is full, ARRPJ flushes tuples from memory. The tuples are discarded instead of being flushed to disk partitions.

8.3.3 Prob

The *PROB*[DGR03, DGR05] approximate join is an instantiation of $DP_X(1)$. The goal of *PROB* is to maximize the quantity of results produced. It assigns a priority to each tuple that arrives. *Prob* can make use of either a fixed or variable memory allocation to store tuples from each of the data streams. For fixed allocation, two priority queues are used, one for each of the data streams. For variable allocation, a single priority queue is used for both streams. The priority for a tuple is determined by the arrival probabilities of the partner stream. We describe how *Prob* works. Given two streams S_1 and S_2, a memory size M. Two priority queues, PQ_1 and PQ_2, (one for each stream) are created. Using a fixed memory allocation, the size of each priority queue is $\frac{M}{2}$. In order to deliver results progressively, a probe-and-insert paradigm is used. When a tuple t_{S1} arrives, it needs to probe all the tuples in the PQ_2 in order to determine join matches. Similarly, when a tuple t_{S2} arrives, it needs to probe all the tuples in PQ_1 for join matches. We consider the case: at time τ, $|S_1|$ and $|S_2|$ tuples have arrived for S_1 and S_2 respectively. Using a variable memory allocation scheme, the size of the single priority queue is M. Whenever tuples arrive from either stream, it will have to scan all the tuples in the priority queue. The time complexity for both the fixed and variable memory allocation is given by O(M($|S_1|$ + $|S_2|$)).

8.3.4 ProbHash

In order to reduce the need to probe all in-memory tuples, we propose a progressive join algorithm, *ProbHash*. *ProbHash* relies on hash partitions to organize the in-memory tuples. In essence, *ProbHash* is a CPU-efficient extension of *Prob* [DGR03, DGR05].

ProbHash organizes the in-memory tuples for each stream by storing the tuples using p priority queues, instead of a single priority queue. The value of p is dependent on the hash function used. The tuples in each priority queue are organized based on a ascending priority order. We denote the set of priority queue for data stream S_i as PQ_{S_i} ($1 \leq$ i ≤ 2). Figure 8.1 shows the two sets of priority queues. Whenever a tuple t_{S1} arrives, its hash value is computed by the hash function (denoted by $\oplus$). It is then used to probe one of the priority queues in PQ_{S_2}. If join matches are found, the result is delivered to the user. Afterwhich, t_{S1} is inserted to one of the priority queues of PQ_{S_1}. The set of priority queues, PQ_{S_1} and PQ_{S_2}, are each allocated $\frac{M}{2}$ memory. Within each priority queue set, we make use of a variable memory allocation scheme which allows the size of the priority queues to grow or shrink dynamically. This mitigates the effect of skewed data distribution, and ensure that the memory can be better utilized. Suppose the average length of each priority queue is L (L $<<$ M), the time complexity for *ProbHash* is given by O(L($|S_1|$ + $|S_2|$)).

When memory is full ($|S_1| + |S_2| =$ M), and a new tuple arrives, we will need to select a tuple to be discarded from amongst the $2p$ priority queues. We first identify the priority queue PQ_i ($1 \leq$ i $\leq 2p$) which contains the tuple with the smallest

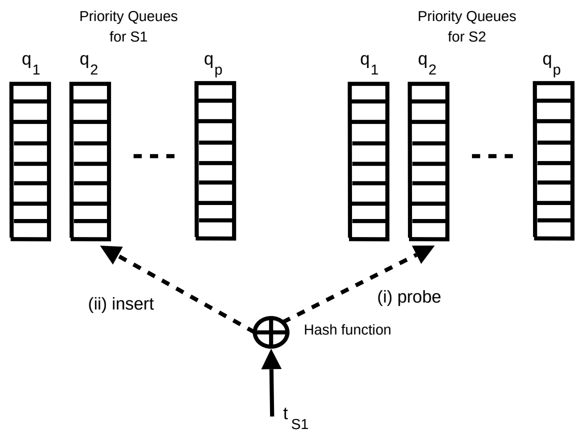


Figure 8.1: Priority Queue for S_1

priority value. The complexity for finding the queue which contains a tuple with the smallest priority value is given by *O(p)*. This is because we only need to scan the first element of each of the $2p$ priority queues. In the case of a tie (i.e several queues with tuples having the smallest priority value), we randomly pick a tuple from one of these queues. Other methods can be used too (e.g. the tuple's age and preferring tuples that are older). We dequeue the tuple with lowest priority. We then compute the hash value for the newly arrived tuple, which is used to determine the priority queue it is inserted into. Due to the variable memory allocation, it is important to note that the size of all the priority queues are not fixed. Hence, if the data distribution is skewed, some priority queues will be longer.

8.3.5 Reservoir Approximate Join (RAJ)

Conventional reservoir sampling [Vit85] is used to produce a fixed size random sample of data. Algorithm 6 describes the details. While data is arriving (line 2), we get the next tuple from the data stream S (line 3). n denotes the total number of tuples that have arrived so far. If the number of tuples in the reservoir is less than the reservoir size $|R|$, we insert the tuple into the reservoir (line 5 to 6). Otherwise, the tuple is inserted into the reservoir with probability $\frac{|R|}{n}$ (line 8 to 10).

Conventional reservoir sampling can also be used in a progressive approximate

Algorithm 6 Conventional Reservoir Sampling

```
1: n = 0
2: while ( !endOfStream(S) ) do
3:     Tuple t = getNextTuple(S)
4:     n = n + 1
5:     if ( n < $|R|$ ) then
6:         Insert t into R
7:     else
8:         Randomly generate a number $\rho$ between 1 to n
9:         if ( $\rho < |R|$ ) then
10:            Replace the $\rho$-th tuple in R with t
11:        end if
12:    end if
13: end while
```

join. We call this the Reservoir Approximate Join (RAJ). This is illustrated in Figure 8.2. Given two streams S_1 and S_2, and memory with size M. Two reservoirs, $Reservoir_{S1}$ and $Reservoir_{S2}$ are created. Each reservoir is allocated $\frac{M}{2}$ memory. For each reservoir, the conventional reservoir sampling technique is used to manage the reservoir. When a tuple t_{S1} arrives, it is used to probe $Reservoir_{S2}$. Results (if any) are produced. Afterwhich, t_{S1} is inserted into $Reservoir_{S1}$. The problem with this approach is that the entire reservoir needs to be scanned in order to find tuples which can be joined with the newly arrived tuple.

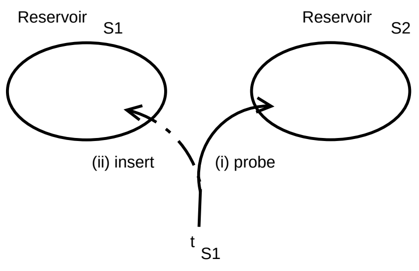


Figure 8.2: Reservoir Approximate Join

8.3.6 Stratified Reservoirs Approximate Join (RAJHash)

In statistics, stratified sampling [Coc77] is another effective technique for sampling from a population. In stratified sampling, the population is divided into disjoint k sub-populations of sizes N_1, N_2,...,N_k respectively. Each sub-population is called a stratum, and is mutually exclusive (i.e. every element in the population must be assigned to only one stratum). Hashing is an effective way to assign each element to exactly one stratum. In order to reduce the need to scan the entire reservoir during probing, we adopt the idea of stratified sampling to organize the reservoir for each stream into multiple sub-reservoirs. We refer to the method where we make use of stratified reservoirs as *RAJHash*. In *RAJHash*, a partitioning scheme is used to organize the tuples in each of the reservoir. The partitioning scheme effectively organizes the tuples into sub-population. In *RAJHash*, this corresponds to the sub-reservoirs that are maintained. We call this algorithm the Stratified Reservoirs Approximate Join *RAJHash*.

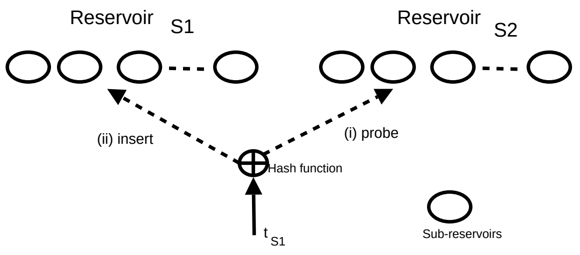


Figure 8.3: Progressive Approximate Join using Stratified Reservoirs

In the stratified reservoir approach, we allocate $\frac{M}{2}$ memory to each reservoir. Each reservoir consists of k sub-reservoirs. For each reservoir, a variable memory allocation scheme is used to allocate memory for the sub-reservoirs. Given a tuple t, the hash function, f(t) = t.value mod k, is used to assign the tuple to one of the sub-reservoirs. t.value denotes the value of the join attribute. Algorithm 7 describes the insertion of a newly-arrived tuple using the stratified reservoir. In Line 1, h denotes the hashed value of the tuple. If n is less than $|R|$, then we will just add the tuple to the *h-th* sub-reservoir (Line 4). If n is greater or equal to $|R|$, then we will need to determine whether to replace a tuple from the reservoir with the newly arrived tuple (Line 6-10). To do this, a random number, ρ (between 1 to n) is generated. If ρ is greater than $|R|$, we discard t. Otherwise, t is used to replace a tuple from the *h-th* sub-reservoir. In this case, even though ρ is less than $|R|$, ρ can be greater than the size of the *h-th* sub-reservoir. To find the tuple to be replaced, we compute i = ρ mod S (where S is the size of the *h-th* sub-reservoir). We then replace the *i-th* tuple in the *h-th* sub-reservoir.

As an implementation optimization, Algorithm 7 first chooses the sub-reservoir using the hash function, and then replace a random tuple in the specific sub-reservoir. It is important to note that in order to maintain a simple random sample for each of the reservoirs, the decision on the tuple to be replaced should not be restricted to just a single reservoir. Instead, a random tuple from any of the sub-reservoirs can be replaced.

Algorithm 7 Stratified Reservoir - Inserting a tuple

```
1: h = f(t)
2: n = n + 1
3: if ( $n < |R|$ ) then
4:    Insert t into the h-th sub-reservoir
5: else
6:    Randomly generate a number $\rho$
      between 1 to $n$ (inclusive)
7:    if ( $\rho < |R|$ ) then
8:       S = Get the size of the h-th sub-reservoir
9:       $i = \rho$ mod S
10:      Replace the i-th tuple with t
11:   else
12:      Discard t
13:   end if
14: end if
```

RAJHash introduces some advantages over *RAJ*. Firstly, it is more CPU-efficient as it reduces the number of in-memory tuples that are probed to identify join matches. Secondly, even in the presence of a skewed distribution, it is able to gracefully allocate more memory for sub-reservoirs which need a large sample, and less memory for sub-reservoirs which contain the skewed values. This is due to the variable memory allocation for the sub-reservoirs. We empirically verify this in Section 8.4.3.

Example

In this example, we illustrate how Stratified Reservoir works. Given two streams S_1={10, 22, 34, 11, 30, 90, 2, 1, 13, 10} and S_2={ 10, 48, 20, 35, 12, 58, 67, 71, 44, 83 }. In this example, the size of the memory M = 10 tuples. Two reservoirs $Reservoir_{S1}$ and $Reservoir_{S2}$ are created for S_1 and S_2 respectively. Each reservoir can hold 5 tuples. In addition, each reservoir is allocated 10 sub-reservoirs. The hash function *f(t) = t.value mod 10* is used to allocate a tuple to one of the 10 sub-reservoirs. We denote a sub-reservoir for stream S_i as $reservoir_i^j$ ($0 \leq$ j $<$ 9) respectively.

For stream S_1, the first tuple arrives. This is inserted into $reservoir_1^0$. Next, a tuple from S_2 arrives. This is first used to probe $Reservoir_{S1}$, which in turn redirects it to sub-reservoir $reservoir_1^0$ which produces a result. After 5 tuples have arrived from each of the data streams, we have the following $reservoir_1^0$ = {10, 30}, $reservoir_1^1$ = {11}, $reservoir_1^2$ = {22}, $reservoir_1^4$ = {34}, $reservoir_2^0$ = {10, 20}, $reservoir_2^2$ = {12}, $reservoir_2^5$ = {35} and $reservoir_2^8$ = {48},. When the sixth tuple from S_1 arrives, $Reservoir_{S1}$ is full. We need to decide whether to discard a tuple from $Reservoir_{S1}$. First, we compute the hash value of the sixth tuple to be 0 (i.e. 90 mod 10). To determine whether to discard the tuple, we randomly generate a number ρ between 1 to 6 (inclusive). If $\rho \leq 5$, then we will replace a tuple in the sub-reservoir $reservoir_1^0$ with this newly arrived tuple. Suppose the value of ρ is 4. It is important to note that there are only two tuples in $reservoir_1^0$. To determine which tuple to be replaced, we compute 4 mod 2 = 0. Thus, the first tuple (value=10) is then replaced with the newly arrived tuple. Thus, sub-reservoir $reservoir_1^0$ = {90, 30}. Similarly, when the sixth tuple (value = 58) from S_2 arrives, we need to decide whether to discard or replace a tuple from $reservoir_2^8$. We generate a random number, ρ between 1 to 6 (inclusive). Suppose rho = 6. Thus, we discard the newly arrive tuple. Thus, sub-reservoir $reservoir_2^8$ = {48}.

8.3.7 Discussion

Approx-RRPJ, *Prob* and *ProbHash* attempt to maximize the quantity (i.e. number of results produced) by sacrificing tuples that do not produce or produce few results. Therefore, they tend to favour results in certain ranges. In contrast, *RAJ* and *RAJHash* strive to maintain a good representative sample. With limited memory, an approximate join algorithm need to effectively make use of the available memory, balancing between quantity and quality of the results produced.

8.4 Performance Evaluation

In this section, we perform an extensive performance study, using both synthetic and real-life datasets. 5 algorithms are used in the performance study: (1) *ARRPJ* (2) *Prob* (3) *ProbHash* (4) *RAJ* and (5) *RAJHash*. We implemented all the progressive approximate algorithms in C++, and conduct the experiments on a Pentium 4 2.4 Ghz PC (1GB RAM).

We evaluate the approximation performance by: (1) Visualizing the quality of the results using normalized result histograms, (2) Measuring the percentage of results (Quantity) and ,(3) Measuring the JS Divergence (Quality). We have also conducted experiments to measure the quality of the results using MSE. As the results using MSE show similar trends to JS Divergence, the results are omitted. We also studied the effects of varying memory sizes. In addition, we also studied the throughput and quality throughput of the various progressive approximate join algorithms by taking snapshots of the result distribution at different time epochs.

The experiment parameters are given in Table 8.1. When the memory allocated to the approximate join is 100%, all tuples fit in memory. Hence, the complete set of join results are produced. We refer to this method as *EXACT*, which we use as a benchmark for comparison for result quality and computing the percentage of results produced by each method.

Table 8.1: Experiment Parameters

Parameter	Values
Memory allocated to approximate join, M	Varies between 10% to 100%
Datasets (DS, Dataset Size)	Skewed (100,000 tuples)
	Extreme (100,000 tuples)
	Real-life: Weather (2,074,948 tuples)

8.4.1 Effect of Skewed Distribution

In this experiment, we investigate the performance of the various methods in the presence of a skewed distribution. The skewed distribution is generated as follows: The frequency of the join attribute values is determined by a Zipfian distribution. The skewness of the Zipfian distribution is determined by a factor η. We set η to be 1.0. We vary the memory allocated to be 10% to 100% of the dataset size (100,000 tuples). The domain of the join attribute value is set to be 1-50.

Approximation Performance

We first study the performance of the algorithm w.r.t to approximation only. Therefore, we consider bounded input streams, and look at the quantity and quality after all the tuples have been processed.

The goal of the first experiment is to visualize the quality of the results produced by the various progressive approximate join algorithms. We fixed the memory allocated to the approximate join to be 10% of the dataset. To achieve this, we plot the result histograms for each of the algorithms. In the y-axis, we show the normalized frequency for each join attribute value. Given the number of results produced by an approximate join method J is $|J|$. The number of results with join attribute value v is given by $|v|$. The normalized frequency is defined as $\frac{|v|}{|J|}$. In the x-axis, we plot the value of the join attributes.

From Figure 8.4(a)-(c), we can observe that *ARRPJ*, *Prob*, and *ProbHash* favor the production of the most likely results. Hence, the results that are produced are skewed towards the join attribute values that appear more frequently. From Figure 8.4(e) and (f), it is visually striking that the normalized histograms for *RAJ* and *RAJHash* are almost identical to the distribution of the complete results (Figure 8.4(a)). For a quantitative comparison of the result quality, we also show the JS Divergence for the various algorithms in Figure 8.4(g). Thus, we can conclude the quality of results produced by *RAJ* and *RAJHash* is higher than that produced by *ARRPJ*, *Prob*, and *ProbHash*.

In the second experiment, we vary the amount of memory allocated to the progressive approximate join. The x-axis shows the amount of memory allocated as a percentage of the total dataset size. The y-axis shows the percentage of results produced and the JS Divergence respectively for Figure 8.4(h) and (i). From the figures, we can observe that the quantity and quality improves as the amount of memory allocated increases. In addition, it is consistently observed in all the algorithms. From Figure 8.4(h), we can observe that the number of results produced by *RRPJ*, *Prob* and *ProbHash* is significantly more than *RAJ* and *RAJHash*. However, from Figure 8.4(i), we can observe that the JS-divergence of *RAJ* and *RAJHash* is much lower. As noted in Section 8.3.7, with limited memory, there is always a tradeoff between quantity and quality.

Throughput and Quality Throughput

Next, we measure the quantity and quality of the results over time. We set the amount of memory allocated to the progressive approximate join algorithms to be 10% of the dataset size. We measure the percentage of complete results produced and the JS Divergence over time (x-axis).

From the results presented in Figure 8.5(a) and (b), we can observe that the throughput of *ARRPJ*, *ProbHash*, and *RAJHash* is significantly better than *RAJ* and *Prob*. However, they produce a lesser percentage of the complete results compared to *Prob*. In Figure 8.5(c), we can observe that the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* is significantly higher than *RAJ* and *RAJHash*. In addition, as time progresses, the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* increases. In contrast, from Figure 8.5(d), we can observe that the JS Divergence of *RAJ* and *RAJHash* decreases with time. This is because the former three methods aims to maximize quantity. Over an extended period of time, this affects the quality of the results. For the sampling-based algorithms, *RAJ* and *RAJHash*, as time progresses, the quality improves. Hence, the decreasing JS Divergence.

ProbHash has significantly better throughput, compared to *Prob*, due to the partitioning of the data space into multiple priority queues. This reduces the number of scans for join matches. In contrast, for *Prob*, a newly arrived tuple will have to scan all the tuples in the corresponding priority queue, which is inefficient. A similar observation can be made between *RAJHash* and *RAJ*. Most importantly, this is achieved without sacrificing the overall result quality over time.

We also studied the tradeoffs between quantity and quality. The results are presented in Figure 8.5(e). As observed in earlier graphs, when the percentage of results increases, the JS Divergence for *ARRPJ*, *ProbHash*, and *RAJHash* increases. In contrast, from Figure 8.5(f), we can observe that for *RAJ* and *RAJHash* the JS Divergence decreases with increasing number of results produced.

8.4.2 Real Life Dataset

In this experiment, we investigate the performance of the various methods using a real-world dataset, consisting of weather data [HWL96].

The dataset consists of monthly cloud measurements, collected by sensors globally. Similar to [DGR03], we chose the data collected for September 1985 and September 1986 as the inputs to the approximate equijoin. The total size of both datasets is approximately 2 million tuples. For each of the dataset, we extracted the values of the latitude and longitude attributes. These attributes denotes the location of sensors which capture the sensors reading. Next, we partition the data universe using a 18 x 36 square grid. Each grid cell is assigned a unique identifier. Each tuple in the dataset, described by its latitude and longitude, is then assigned the value of the unique identifier. We then perform an equijoin between the 1985 and 1986 datasets.

We omit the results for *Prob* and *RAJ*, and show only their more efficient counterparts, *ProbHash* and *RAJHash* respectively.

Approximation Performance

Similar to Section 8.4.1, we first study the performance of the algorithm w.r.t to approximation only. Thus, we consider bounded input streams, and look at the quantity and quality after all the tuples have been processed.

In the first experiment, we fixed the memory allocated to the approximate join to be 10% of the dataset. We present the result histograms for the various approximate join algorithms in Figure 8.6(a)-(d), where we can observe the quality of the result distribution. We omit the result histograms for *Prob* and *Reservoir* as they exhibit similar trends to *ProbHash* and *SReservoir* respectively. From Figure 8.6(a)-(d), we can observe that the normalized result histograms for *RAJHash* is similar to *Exact*. In contrast, we can observe that *ARRPJ* and *ProbHash* indeed maximize the result quantity for the join attribute values which appear more frequently.

In the second experiment, we vary the amount of memory allocated for the progressive approximate join algorithms. The results are presented in Figure 8.6(e) and (f). From the figures, we can also observe that the quantity and quality improves as the amount of memory allocated increases. Also, we can observe in Figure Figure 8.6(e) that the number of results produced by *RRPJ*, *Prob* and *ProbHash* is significantly more than *RAJ* and *RAJHash*. From Figure 8.6(f), we can observe that the JS-divergence of *RAJ* and *RAJHash* is much lower.

Throughput and Quality Throughput

Next, we measure the quantity and quality of the results over time. We set the amount of memory allocated to the progressive approximate join algorithms to be 10% of the dataset size. We measure the percentage of complete results produced and the JS Divergence over time (x-axis).

From Figure 8.7(a) and (b), we can observe that the throughput of *ARRPJ*, *ProbHash* and *RAJHash* is significantly better than *RAJ* and *Prob*. This is similar to the observation made for the skewed synthetic dataset. From Figure 8.7(c), we can observe that the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* is significantly higher than *RAJ* and *RAJHash*. As time progresses, the JS Divergence of *ARRPJ*, *ProbHash*, and *Prob* increases. In contrast, from Figure 8.7(d), we can observe that the JS Divergence of *RAJ* and *RAJHash* initially increases. This is due the arrival of non-representative tuples in the beginning. Hence when these tuples are used in the join, the results are not representative (hence the increasing JS divergence at the initial stages). However, when time progresses, the JS Divergence decreases with time. The tradeoffs between quantity and quality are presented in 8.7(e) and (f). We can observe that the JS Divergence increases as the percentage of results produced by *ARRPJ*, *Prob* and *ProbHash* increases. In contrast, the JS Divergence for *RAJ* and *RAJHash* decreases over time. In Figure 8.7(f), the initial increase in JS Divergence is due to the effects discussed earlier for Figure 8.7(d).

8.4.3 Effect of Extreme Dataset

In this experiment, we investigate the performance of the various methods in the presence of the extreme scenario [DGR03, DGR05]. The extreme scenario is characterized by having join attribute values that appear less frequently for each dataset. Figure 2.1 shows the join attributes values used in the extreme scenario for two data streams, R1 and R2. For the experiments, the value of b_1 and b_2 is set to 1 and 2 respectively.

From Figure 8.8(a), we can observe that except for *RAJ*, all methods are able to generate 100% of the results. An interesting observation is that *RAJHash* is able to generate 100% of the results, whereas *RAJ* does not. This is because *RAJHash* is able to keep the rare values b_1 (from R_1) and b_2 (from R_2) in the sub-reservoir. When the tuples with join attribute values b_2 (from R_1) and b_1 (from R_2) arrives, they are assigned to the other sub-reservoir. In this way, *RAJHash* was able to maintain a random uniform sample for each of the sub-reservoirs.

[CMN99, DGR03] noted that using the Reservoir method will not produce any join results for the extreme scenario. The assumption made was that the Reservoir needs to be completely filled for either of the data streams, before join processing can start. In contrast, when a probe-insert paradigm is used to continuously probe the reservoir while it is being built, join results can still be produced since the rare tuples have not been discarded from the reservoir yet. In the experiment, we show that even in the extreme scenario, *RAJ* will still produce results (instead of an empty result set) as it progressive probe the reservoirs for result.

In addition, we also present a softer variant of the extreme scenario. In this variant, we relax the constraints on the appearance of specific join attribute values. In this variant, the values of b_1 (from R_1) and b_2 (from R_2) have a 50% chance of re-appearing in the dataset. From Figure 8.9(a), we can observe that *Prob*, *ProbHash*, *RAJ* and *RAJHash* produce the same percentage of results. From Figure 8.9(b), we can observe that the quality of the results produced by *RAJ* and *RAJHash* are significantly better.

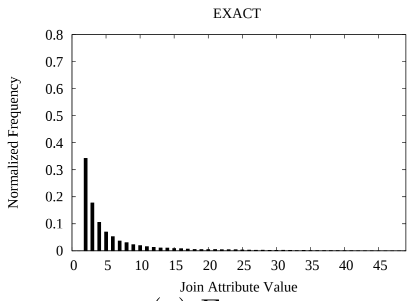
8.5 Summary

In this chapter, we have investigated the problem of progressive approximate join processing using limited memory.

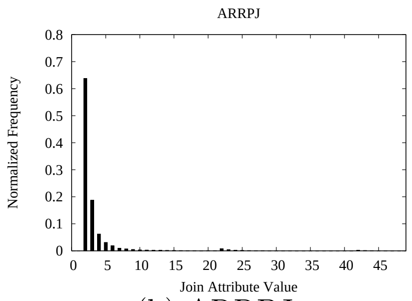
Though several approximate join processing techniques have been proposed, the focus has always been maximization of the size of the set of results. In this work, we have clearly differentiated the notions of quantity and quality. We have shown that algorithms can favor one or the other. We have also empirically demonstrated that there exists a trade-off between the two strategies as they compete for the usage of memory.

We have shown that stratification of memory with hash partitioning can significantly improve the efficiency of progressive approximate joins and therefore improve throughput without sacrificing quantity and quality. We have also shown that reservoir sampling based progressive approximate joins are superior when quality matters.

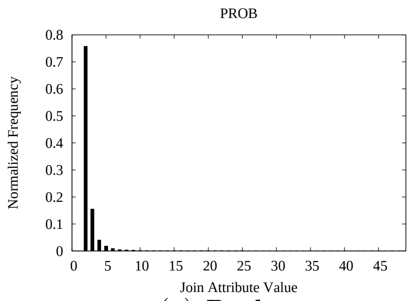
We propose four new progressive approximate join algorithms: *ARRPJ*, *ProbHash*, *RAJ* and *RAJHash*. The former two, like *Prob*, favor quantity, the latter two favor quality. *ProbHash* improves on *Prob* on every aspects. *RAJ* and *RAJHash* produce results of significantly better quality. Interestingly, although they produces less results, *RAJ* and *RAJHash* are the fastest to produce because of the simplicity of the reservoir data structure and algorithm.



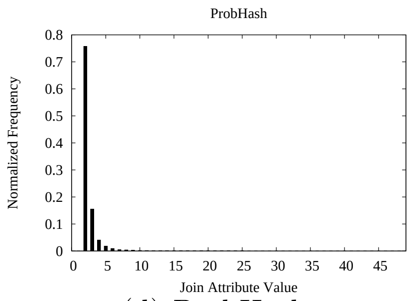
(a) Exact



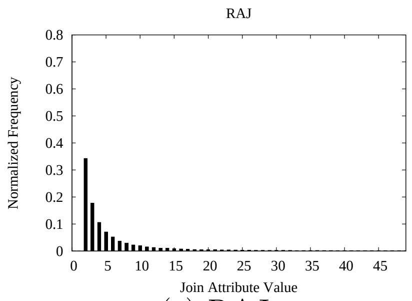
(b) ARRPJ



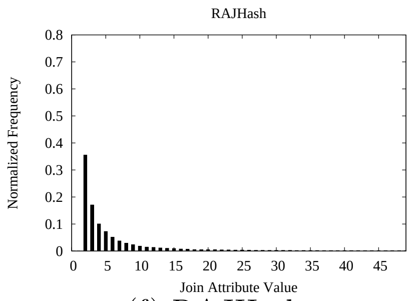
(c) Prob



(d) ProbHash



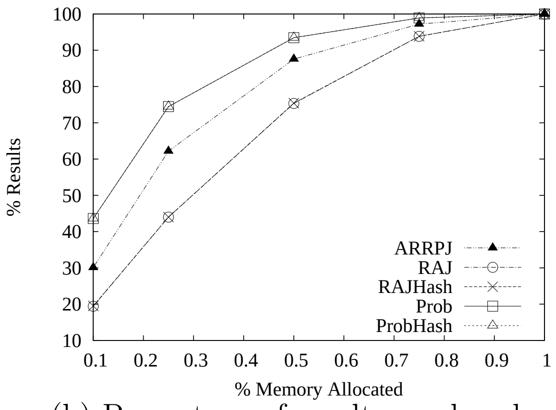
(e) RAJ



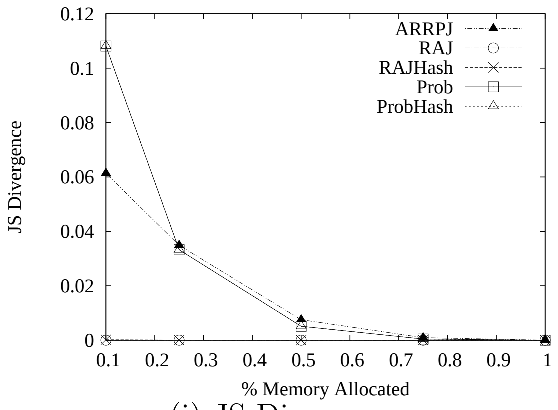
(f) RAJHash

	ARRPJ	Prob	ProbHash	RAJ	RAJHash
JS Divergence	0.06136	0.10812	0.10811	0.00001	0.00018

(g) JS Divergence

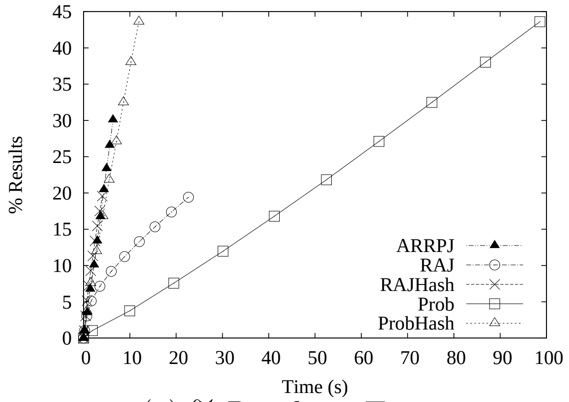


(h) Percentage of results produced

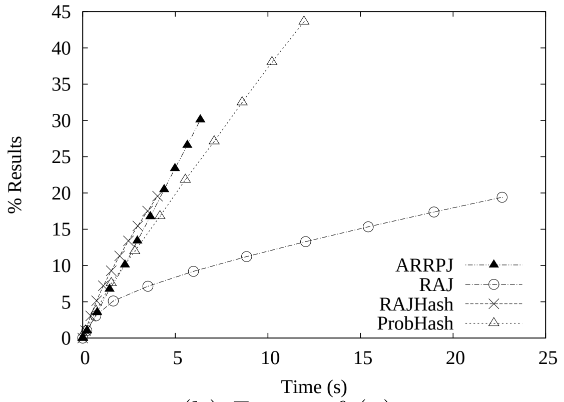


(i) JS Divergence

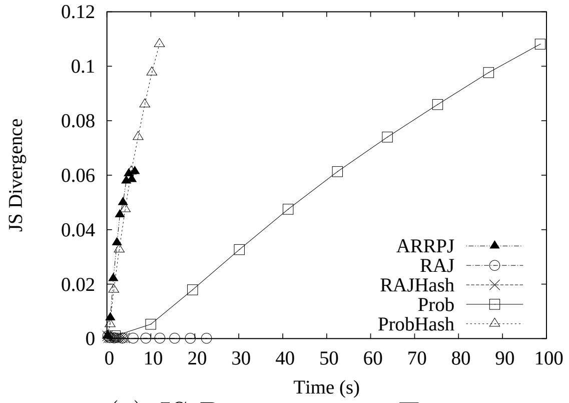
Figure 8.4: Skewed Dataset



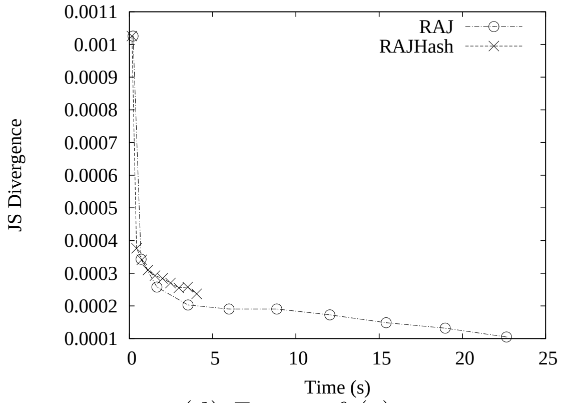
(a) % Result vs Time



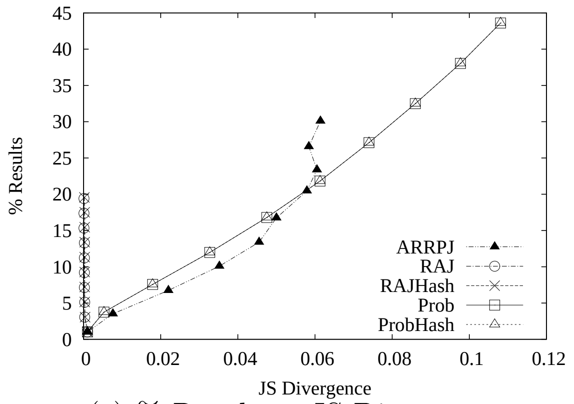
(b) Zoom of (a)



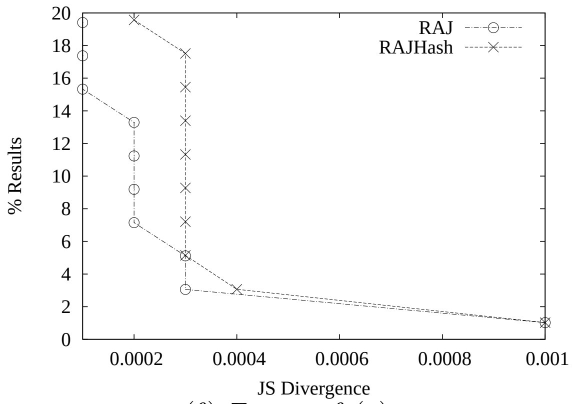
(c) JS Divergence vs Time



(d) Zoom of (c)

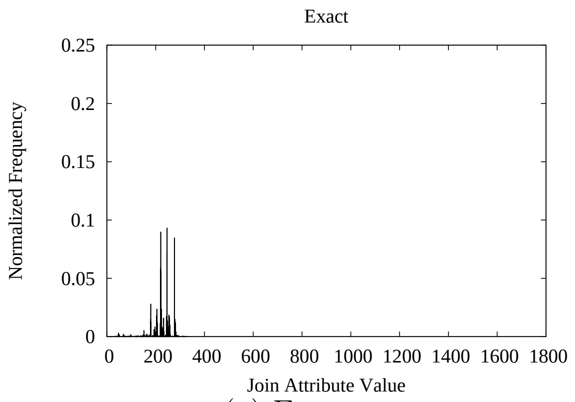


(e) % Result vs JS Divergene

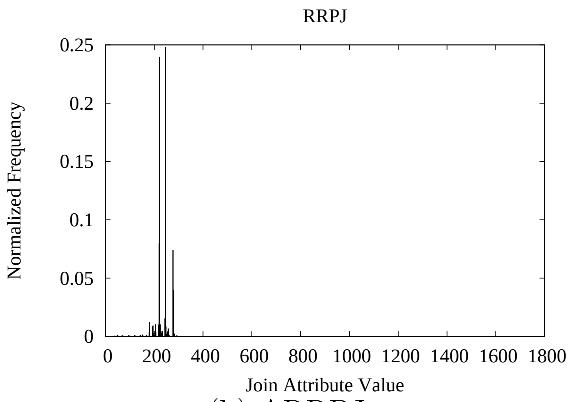


(f) Zoom of (e)

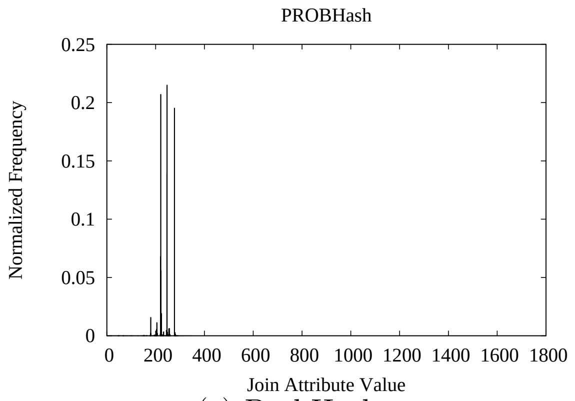
Figure 8.5: Skewed Dataset : Throughput and Quality Throughput



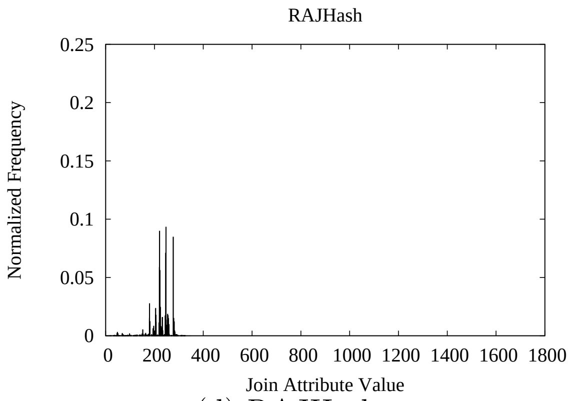
(a) Exact



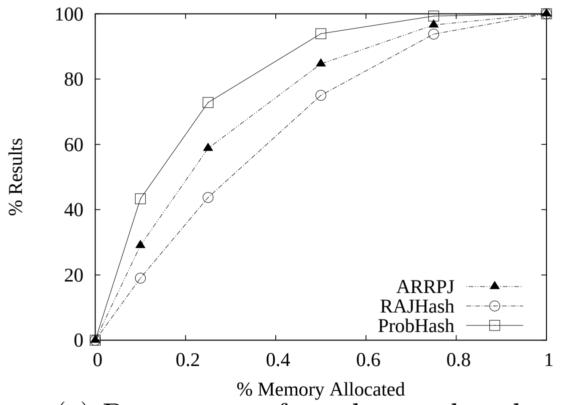
(b) ARRPJ



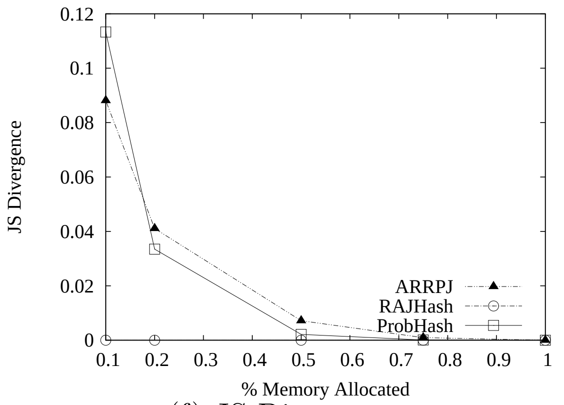
(c) ProbHash



(d) RAJHash

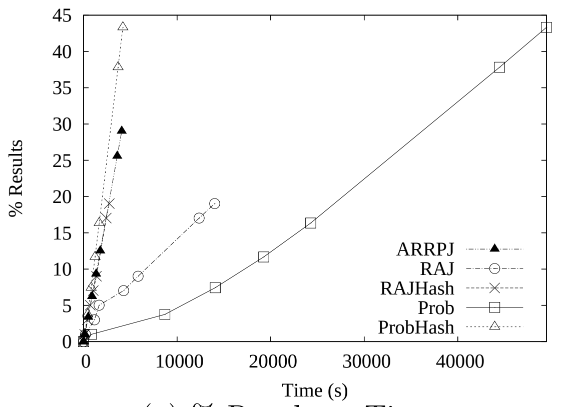


(e) Percentage of results produced

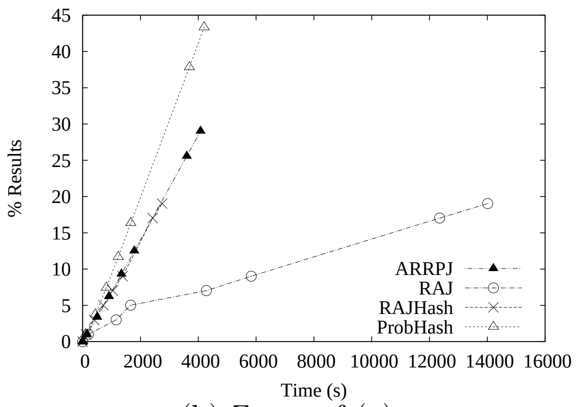


(f) JS Divergence

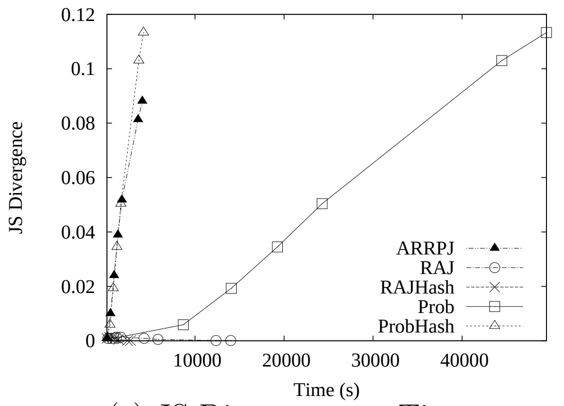
Figure 8.6: Real Life Dataset (WEATHER)



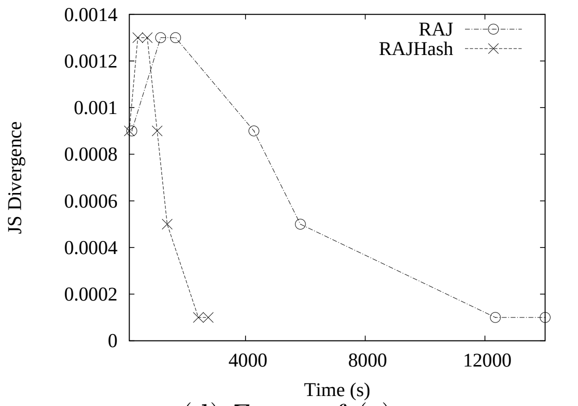
(a) % Result vs Time



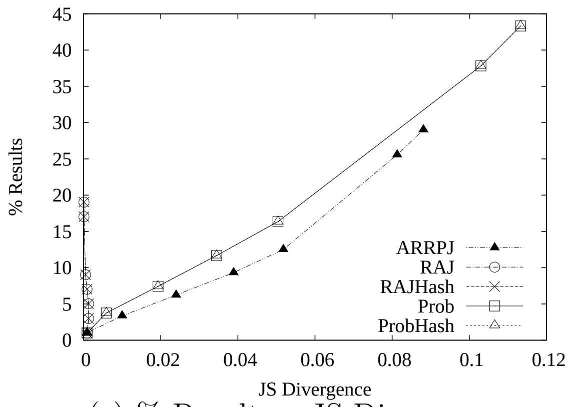
(b) Zoom of (a)



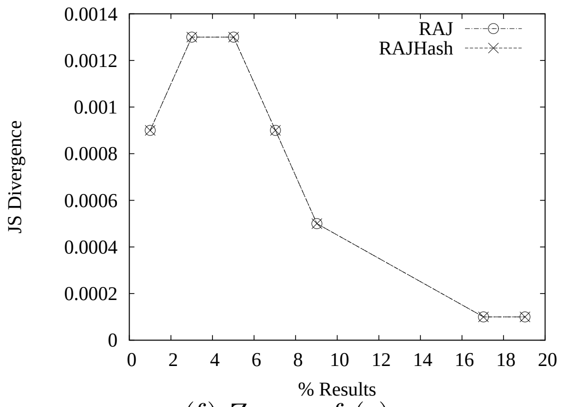
(c) JS Divergence vs Time



(d) Zoom of (c)

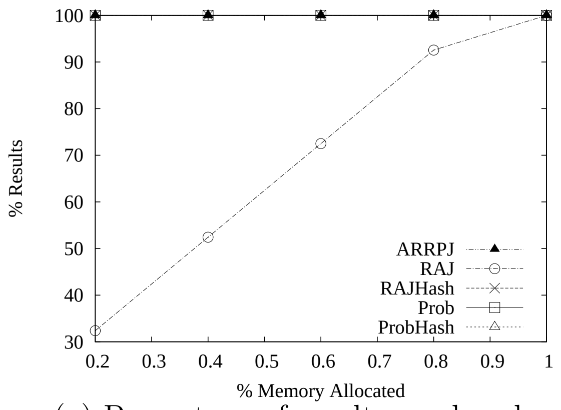


(e) % Result vs JS Divergene

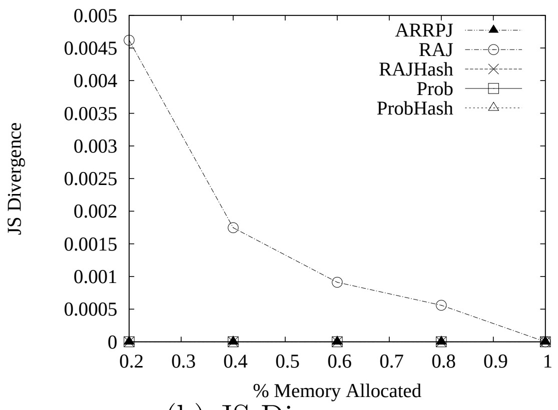


(f) Zoom of (e)

Figure 8.7: Real Life Dataset (WEATHER): Throughput and Quality Throughput

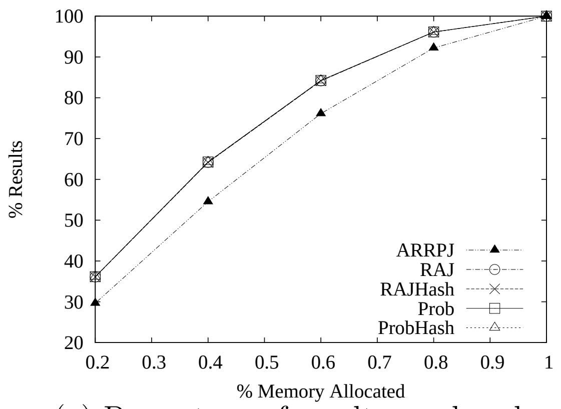


(a) Percentage of results produced

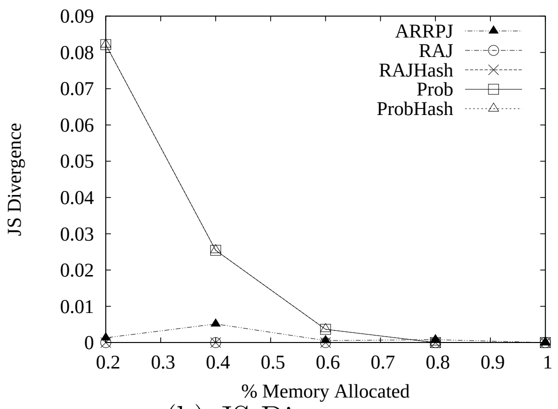


(b) JS Divergence

Figure 8.8: Extreme Scenario : Vary Amount of Memory



(a) Percentage of results produced



(b) JS Divergence

Figure 8.9: Extreme Scenario Variant : Vary Amount of Memory

Chapter 9

Progressive, Approximate Sliding Window Join

9.1 Introduction

In many data stream applications, the sliding window model of data processing is commonly used. Firstly, users are interested in processing the current data (i.e. data that is in the current window), instead of old data (i.e. data that are out of the window). Secondly, due to the unbounded nature of data streams, the data volume is often much larger than the available resources (e.g. memory, and computational resources). Thus, it is more practical to process and answer queries using windows of data, instead of processing the entire data stream.

In sliding window model of data stream processing, two types of of sliding windows are commonly used [BDM02]: sequence-based and timestamp-based windows. In a sequence-based window, the $|W|$ most recent tuples are kept in the window, where $|W|$ refers to the window size. A tuple expires from a sequence-based window when it is no longer one of the $|W|$ most recent tuples. In a timestamp-based window, a tuple is assigned an arrival timestamp when it is first added to the window. A tuple expires from a timestamp-based window when its timestamp is no longer within the valid time interval of a window.

A synopsis is a key building block for progressive, approximate algorithms over data streams. The synopsis provides an approximate representation for all the tuples of the data stream. Many different types of synopsis have been proposed. These includes: wavelets, sketches, samples and histograms. Most synopsis are designed for providing estimations to aggregation queries, and seldom extends to providing approximate results for join queries. In addition, the authors of [DGR03] have even disqualified sampling based methods based on the extreme scenario given in [CMN99] without further experiments. We have shown in [TBL08a] that this disqualification is mistaken, and that a stratified sampling approach can be used to effectively and efficiently provide approximate results for join queries. In [TBL08a], a stratified sample is maintained for each of the data streams. Whenever a new tuple from one stream arrives, it is first used to probe (for matching tuples that can be joined based on the join predicate) the stratified sample for the other data stream. Afterwhich, the tuple is then inserted into its own stratified sample. While [GLH06] provides a systematic study on the maintenance of a random sample, the focus was on handling arbitrary insertions/deletions, and dynamically resizing of the sample size. In addition, the techniques proposed do not show how they can be directly extended for sliding window joins. Hence, we did not include it in our comparison.

In [TBL08a], the stratified sample is maintained over the entire data stream, and do not consider a sliding window model. In order to provide quality results to various types of queries in a sliding window model, it is important that the samples that are maintained take into account the sliding window semantics. The samples must be representative of the data in the sliding window, and not the entire data stream. In sampling theory [Coc77], the sample is referred to as a simple, random sample (srs) of the sliding window. Our work builds on the initial work by Brian et.al [BDM02], which considered the problem of sampling from a moving window over a single data stream. While Brian et.al [BDM02] presents a discussion on the various issues that need to be considered for sampling over data streams, they do not provide empirical evidence.

In this chapter, we propose a generic framework for designing sampling-based progressive sliding window joins. In order to evaluate the effectiveness of various sampling techniques we considered the use of four sliding-window based sampling techniques. These includes: *Expire* [BDM02], and 2 new sliding window sampling algorithms: *FIFO* and *WinRes*. As a baseline, we also included the conventional reservoir sampling. In order to study the effectiveness of each of these sampling techniques, we study the performance of each of the techniques prior to incorporating them within the sliding window join framework. We present both empirical and theoretical analysis for each of the proposed sampling techniques. Next, we incorporated each of these sampling techniques in the sliding window join framework, and conduct an extensive performance evaluation.

The chapter is organized as follows. In Section 9.2, we present the framework for a sampling-based progressive sliding window join. In Section 9.3, we present four sampling techniques. We conduct an extensive performance evaluation in Section 9.4.

9.2 Progressive Sliding Window Join

In this section, we propose a sampling-based sliding window join framework.

Given two streams R_1(A,B) and R_2(B,C), where A, B and C are attributes of the data streams. Let the *i-th* tuple from R_1 and the *j-th* tuple from R_2 be denoted by $t_{R1}(a_i,b_i)$ and $t_{R2}(b_j,c_j)$ respectively. The size of a sequence-based sliding window W is given by $|W|$. In the sampling-based approach, a sample is maintained for each of the data streams. Whenever a tuple t from one data stream arrives, it is used to probe the sample for the other data stream for results.

In Algorithm 8, two samples S_{R1} and S_{R2} are maintained for data streams R_1 and R_2 respectively. The size of each of the sample is given by $|S_{R1}|$ and $|S_{R2}|$. Various sampling techniques (described in Section 9.3) can be used.

The probe step is given in Algorithn 9. During the probing for join matches, Algorithn 9 checks for expired tuples in the sample (Line 8) using *hasTupleExpired()*. A tuple has expired when it is no longer in the sliding window. This is because depending on the sampling technique used, the sample might contain expired tuples. The check prevents erroneous results from being produced when a newly arrived tuple is joined with an expired tuple.

Algorithm 8 Sampling-Based Sliding Window Join Framework

```
1: while ( !endOfStreams($R_1$,$R_2$) ) do
2:    tuple $t$ = select($R_1$,$R_2$)
3:    if ( t.src == $R_1$) then
4:       $S_{R2}$.probe(t)
5:       $S_{R1}$.insert(t)
6:    else if (t.src == $R_2$) then
7:       $S_{R1}$.probe(t)
8:       $S_{R2}$.insert(t)
9:    end if
10: end while
```

Algorithm 9 Sampling-Based Sliding Window Join - Probe

```
1: Let $p$ denote the tuple that is used to probe the sample
2: Let $p_{value}$ and $t_{value}$ denote the attribute value
3: for tuples $p$ and $t$ respectively. The attribute used is
4: defined by the join predicate.
5: Let $R$ denote the result set
6: $R = \{\}$
7: for ( each $t$ in sample $S$ ) do
8:    if (!hasTupleExpired($t$)) then
9:       // Tuple has not expired
10:      if ( $p_{value} == t_{value}$) then
11:         Add $(p,t)$ to $R$
12:      end if
13:   end if
14: end for
15: return $R$
```

9.3 Sliding Window Sampling

In this section, we consider various ways to perform sliding window sampling.

We consider the problem of maintaining a simple random sample S, on a window W. W is a sequence-based sliding window over a data stream D. The size of S, W are $|S|$ and $|W|$ respectively ($|S| <= |W|$). n denotes the number of tuples that have arrived so far.

9.3.1 Reservoir

Reservoir sampling maintains an unbiased sample of $|S|$ tuples in a data stream. Assume that n tuples have arrived. When $n \leq |S|$, then the tuple is added to the reservoir (i.e. sample). When $n > |S|$, the reservoir sampling technique selects a tuple to be replaced. This is achieved by randomly generating a value,ρ, between 1 to n. If $\rho > |S|$, then the newly-arrived tuple is discarded. Else, the tuple replaces the ρ-*th* tuple in the sample. It is shown in [MB83, Agg07] that the reservoir sampling technique maintains an unbiased simple random sample at any point in time.

Algorithm 10 Conventional Reservoir Sampling

```
n = 0
while ( !endOfStream() ) do
    Tuple t = getNextTuple()
    n = n + 1
    if ( n < $|S|$ ) then
        Insert t into R
    else
        Randomly generate a number $\rho$ between 1 to n
        if ( $\rho < |S|$ ) then
            Replace the $\rho$-th tuple in $S$ with t
        end if
    end if
end while
```

9.3.2 FIFO

First-in-First Out window sampling (*Fifo*) maintains a sample of $|S|$ tuples. Fifo maintains the sample as a queue of tuples. Assume that n tuples have arrived. When $n \leq |S|$, then the tuple is added to the sample. When $n > |S|$, *Fifo* determines whether a replacement should be made. This is achieved by randomly generating a value,ρ, between 1 to n. If $\rho < |S|$, then the earliest tuple that have arrived is dequeued, and the new tuple enqueued.

Consider the following example. Given that $D = \{1, 5, 2, 3, 18, 9, 6, 10, ...\}$, $|W|$ = 3, and $|S| = 2$. Let t_v denote the v-th tuple in D. After t_1 and t_2 have arrived, S

= {1,5}. When t_3 arrives, the sample is full. Hence, we need to determine whether t_3 is included in the sample. A random number, ρ is generated between 1 to 3. If ρ is greater than 3, then the newly arrived tuple is discarded. In this case, suppose ρ = 2, we remove the earliest tuple (i.e. in FIFO manner) in the sample. This is replaced with the newly arrived t_3. Thus, S = {5,2} and W = {1,5,2}.

Algorithm 11 FIFO

```
1: n = 0
2: while ( !endOfStream() ) do
3:    Tuple t = getNextTuple()
4:    n = n + 1
5:    if ( n < $|S|$ ) then
6:       Insert t into S
7:    else
8:       Randomly generate a number $\rho$ between 1 to n
9:       if ( $\rho < |S|$ ) then
10:         Remove the first tuple in the sample
11:         Insert $t$ as the last tuple in the sample
12:      end if
13:   end if
14: end while
```

Analysis

Let *pos(t)* denote the position of an arbitrary tuple t in the *FIFO* sample. *pos(t)* = 1 if t is the first tuple in the *FIFO* sample, and 1 < *pos(t)* ≤ $|S|$ otherwise. Given any arbitrary tuple t in the *FIFO* sample, let $P_{removed}$(t) denote the probability that the tuple t is removed from the *FIFO* sample when $\rho < |S|$ (i.e. a tuple needs to be removed from the sample).

$$\mathrm{P}_{removed}(t) = \begin{cases} 1 & pos(t) = 1 \\ 0 & pos(t) > 1 \end{cases} \tag{9.1}$$

We wish to compute P_k, the probability that any particular sample is chosen at the *k-th* step. For $k \leq |S|$ (the sample is not completed filled), $P_k = 1$. For $k > |S|$, we consider two cases. In the first case, an arbitrary tuple t is replaced by a newly arrived tuple. In the second case, an arbitrary tuple t is discarded. We prove both cases by induction.

In the first case, the probability that a newly arrived tuple is added to the sample is $|S|$/(k+1). For k = $|S|$, assume

$$P_k = 1/\begin{pmatrix} k \\ |S| \end{pmatrix} \tag{9.2}$$

Thus, P_{k+1} is derived as follows:

$$\begin{aligned} &P_{k+1} \\ &= P_k(\tfrac{|S|}{k+1})(1) \\ &= [1/\begin{pmatrix} k \\ |S| \end{pmatrix}](\tfrac{|S|}{k+1})(1) \\ &= \tfrac{|S|!(k-|S|)!|S|}{(k+1)!} \end{aligned} \tag{9.3}$$

In the second case, the probability that a tuple is discarded is given by (1 - $|S|$/(k + 1)). Thus, P_{k+1} is derived as follows:

$$\begin{aligned} &P_{k+1} \\ &= P_k(1 - |S|/(k+1)) \\ &= [1/\begin{pmatrix} k \\ |S| \end{pmatrix}](1 - |S|/(k+1)) \\ &= \tfrac{|S|!(k+1-|S|)!}{(k+1)!} \\ &= [1/\begin{pmatrix} k+1 \\ |S| \end{pmatrix}] \end{aligned} \tag{9.4}$$

9.3.3 Expired Reservoir Sampling (Expire))

The Expired Reservoir sampling (*Expire*) technique is introduced in [BDM02] as a sampling technique for moving windows of streaming data. The reservoir sampling technique is used to maintain the sample for the first $|W|$ tuples. When $n > |W|$, then *Expire* will determine the tuple that has expired in the sample. This expired tuple is then replaced with a newly arrived tuple.

Consider the following example. Given that $D = \{1, 5, 2, 3, 18, 9, 6, 10, ...\}$, $|W|$ = 3, and $|S| = 2$. Let t_v denote the v-th tuple in D. After t_1 and t_2 have arrived, S = {1,5}. When t_3 arrives, the sample is full. Hence, we need to determine whether t_3 is included in the sample. A random number, ρ is generated between 1 to 3. If ρ is greater than 3, then the newly arrived tuple is discarded. In this case, suppose ρ = 2, the second tuple in S (i.e. 5) is replaced with the newly arrived tuple. Thus, S = {1,2}, and W = {1,5,2}. At this point, the size-3 window moves. W = {5,2,3}. When t_4 arrives, we check whether the sample contains any expired tuples. In the sample, we observed that t_1 has expired. We replace it with t_4. S = {3,2}. W = {2,3,18}. When t_5 arrives, we observe that none of the tuples have expired. Thus, t_5 is discarded. W = {3,18,9}. When t_6 arrives, t_3 (i.e. value 2) expired. Thus, it is replaced with t_6. S = {3,9}. Afterwhich, W = {18,9,6}. When t_7 arrives, t_4 (i.e. value 3) has expired. Thus, it is replaced. S = {6,9}.

As noted in [BDM02], one of the problems with the *Expire* technique is that the sampling technique is periodic in nature. For example, if a *i-th* tuple is included in the sample, then all subsequent *j-th* tuple will also be included in the sample, where $j = i + cn$ ($c > 0$ and $n > 0$).

Analysis

In reservoir sampling [Vit85], the probability of including a newly arrived tuple in the sample is $\frac{s}{n}$, where n is the number of tuples that have arrived. We further show that the reservoir maintained by *Expire*, at any time t, is a simple random sample without replacement.

At any time time, when the window slides, two cases can occur. In the first case, a newly arrived tuple in the window replaces a tuple in the sample. In the second case, the newly arrived tuple is discarded.

In the first case, we consider the scenario where a tuple in the reservoir is replaced with a newly arrived tuple. The probability of adding the newly arrived tuple in the window to the sample is $\frac{s}{w}$. Since an expired tuple needs to be removed from W, and a randomly chosen tuple needs to be replaced in S, the probability of choosing a

Algorithm 12 Expired Reservoir Sampling

```
while ( !endOfStream() ) do
    Tuple t = getNextTuple()
    if ( $n < |S|$ ) then
        Insert t into the $S$
    else
        if ( $n < |W|$ ) then
            // Perform reservoir-style sampling
            $\rho$ = Randomly generate a number between between 1 and $|W|$ (inclusive)
            if ( $\rho < |S|$ ) then
                Replace the $\rho$-th tuple in $S$ with $t$
            end if
        else
            Check whether there are expired tuples in the reservoir
            If there is a tuple that have expired, replace it with $t$
        end if
    end if
end while
```

sample of size *s-1*, from a window of size *w-1* is given by $1/\begin{pmatrix} w-1 \\ s-1 \end{pmatrix}$. Thus, the probability that a tuple in the reservoir is replaced by a newly arrived tuple from W, is given by

$$\frac{s}{w}(1/\begin{pmatrix} w-1 \\ s-1 \end{pmatrix}) = \frac{s}{w}(\frac{(s-1)!(w-1-(s-1))!}{(w-1)!}) \tag{9.5}$$

Simplifying 9.5, we obtain

$$\frac{s}{w}(\frac{(s-1)!(w-s)!}{(w-1)!}) = \frac{s!(w-s)!}{w!} = 1/\begin{pmatrix} w \\ s \end{pmatrix} \tag{9.6}$$

In the second case, we consider the scenario where a newly arrived tuple from W is discarded. The probability that a newly-arrived tuple is discarded is $1 - \frac{s}{w}$. However, it is important to note that as W slides, the oldest tuple in W expires. Thus, the probability of selecting a sample of size s from a window containing *w-1* tuples is $1/\begin{pmatrix} w-1 \\ s \end{pmatrix}$. Thus, the probability of selecting a sample in this scenario is given by

$$(1-\frac{s}{w})(1/\begin{pmatrix} w-1 \\ s \end{pmatrix}) = (\frac{w-s}{w})(\frac{s!(w-s-1)!}{(w-1)!}) \tag{9.7}$$

Simplifying 9.7, we obtain

$$\frac{s!(w-s)!}{w!} = 1/\begin{pmatrix} w \\ s \end{pmatrix} \tag{9.8}$$

In both cases, we have showed that the sample maintained by *Expire* is indeed a simple random sample without replacement.

9.3.4 Comparison with an extreme case

Next, we consider an extreme case. In the extreme case, we always replace a randomly chosen tuple from the reservoir when a new tuple arrived. In this scenario, the probability of selecting a sample of size *s-1* from a window containing *w-1* tuples is given by

$$1/\begin{pmatrix} w-1 \\ s-1 \end{pmatrix} = \frac{(s-1)!(w-1-(s-1))!}{(w-1)!} = \frac{(s-1)!(w-s)!}{(w-1)!} \tag{9.9}$$

From Equation 9.9, we can observe that the extreme case does not guarantee that a simple random sample is obtained. The main difference between the extreme case and the case discussed in the earlier section lies in the ratio $\frac{s}{w}$, which determines the probability of including the newly arrived tuple in the sample.

9.3.5 Windowed Reservoir (*WinRes*)

WinRes is a reservoir-based sampling [Vit85] technique which maintains a random sample over a sliding window. Similar to reservoir sampling, Windowed Reservoir (*WinRes*) maintains a sample of $|S|$ tuples. The main difference between the Reservoir and *WinRes* is that *WinRes* checks for expiration of tuples. Assume that n tuples have arrived. When $n \leq |S|$, then the tuple is added to the reservoir (i.e. sample). When $n > |S|$, *WinRes* first checks whether there are tuples in the sample that have expired. If there are tuples that have expired in the sample, *WinRes* replaces one of the expired tuples with the newly arrived tuple. If there are no expiring tuples, *WinRes* needs to determine whether a replacement should be made. This is achieved by randomly generating a value,ρ, between 1 to n. If $\rho < n$, then the ρ-*th* tuple in the tuple is replaced with the newly arrived tuple.

Consider the following example. Given that D = {1, 5, 2, 3, 18, 9, 6, 10, ...}, $|W|$ = 5, and $|S|$ = 2. Let t_v denotes the v-th tuple in D. At time t = 2, S = {1,5}. At time t = 3, since the reservoir is full, we determine whether the newly-arrived tuple, t_3 (i.e. value 2) , is included in the reservoir by generating a number, ρ between 1 to 3 (inclusive). If $\rho \leq 2$, we will include t_3 in the reservoir. Otherwise, we discard it.

Suppose at time t = 5, W ={1,5,2,3,18} and S = {1,18}. At time t=6, window W slides, and W = {5,2,3,18,9}. In order to decide whether t_6 (i.e. value 9) is included in S, we first determine whether there are any expired tuples in S. Since t_1 has expired, we remove it from S. t_6 is added to S. Thus, S = {9,18}. At time t=7, W = {2,3,18,9,6} and S={9,18}. Both t_5 and t_6 in S are valid tuples w.r.t W. Hence, we cannot discard any of the tuples. As there is no more available space, we have to decide whether to discard the newly arrived tuple t_7 (i.e. value 6). This is determined by generating a random number, ρ, between 1 to 7 (inclusive). If $\rho \leq 6$, t_6 replaces the 6th tuple. Otherwise, it is discarded. Suppose ρ = 7. We discard t_7. Thus, S={9,18}.

9.4 Performance Evaluation

In this section, we perform an extensive performance study, using synthetic datasets, which allow us to control the changes in the data distribution. We implemented the various sliding window sampling-bsaed join algorithms in C++: (1) Fifo, (2) Reservoir (Res), (3) Expire and (4) Window Reservoir (WinRes).

In addition, we also conducted an extensive study of the performance for each

Algorithm 13 Window Reservoir Sampling

```
 1: n = 0
 2: while ( !endOfStream() ) do
 3:    Tuple t = getNextTuple()
 4:    n = n + 1
 5:    if ( n < $|S|$ ) then
 6:       Insert t into S
 7:    else
 8:       Check whether there are expired tuples in the sample
 9:       if ( there is an expiring tuple ) then
10:          Replace it with $t$
11:       else
12:          Randomly generate a number $\rho$ between 1 to $|W|$
13:          if ( $\rho < |S|$ ) then
14:             Replace the $\rho$-th tuple in $S$ with t
15:          end if
16:       end if
17:    end if
18: end while
```

of the sampling methods. This is presented in Appendix D. In the performance evaluation for the sliding window join, we omit the chain sampling method [BDM02], because it cannot guarantee correctness of results as it maintain multiple samples of size 1. As tuples can be duplicated in the multiple samples, it can potentially produce duplicate results.

The synthetic dataset, D, consists of 500000 tuples. The data is distributed equally between two data streams (i.e. Each data stream consist of 250000 tuples). In addition, the distribution of the data changes every 50000 tuples (i.e. or $0.1|D|$). This is achieved by using a zipfian data distribution with factor, ζ. For each $0.1|D|$ of data, we randomly generated a ζ factor between 0.0 and 2.0 (inclusive). In addition, to ensure that the skewed values do not cluster within a fixed value range, we also shifted the value ranges for each $0.1|D|$ of data generated.

The experiments are conducted on a Pentium 4 2.4 Ghz PC (1GB RAM). We evaluate the performance of the sliding window join using different sampling algorithms. The MSE between the sample and the actual result distribution for each window is measured. While we could have make use of the JSD measure used in Chapter 8,

Parameter	Values
Dataset Size, $\|D\|$	500,000
Frequency of Data Distribution Change, f	Every $0.1\|D\|$
Window Size, $\|W\|$	$0.02\|D\|$, $0.04\|D\|$, $0.06\|D\|$, $0.08\|D\|$, $0.1\|D\|$
Sample Size, $\|S\|$	$0.2\|W\|$, $0.4\|W\|$ $0.6\|W\|$, $0.8\|W\|$, $1.0\|W\|$

Table 9.1: Experiment Parameters

we choose to make use of MSE measure (which is easier to compute) in this chapter due to the need to fully automate the computation of the quality of large number of snapshots. This is achieved by taking a snapshots of the sample at regular intervals, and then compare the sample distribution with the distribution of the results within the given window. Unless otherwise stated, the experiment parameters given in Table 9.1 are used.

9.4.1 Progressive Sliding Window Join

In this section, we evaluate the performance of the progressive sliding window join using four sampling techniques. These includes Reservoir (Res), FIFO, Expired Reservoir Sampling (Expire), and Windowed Reservoir (WinRes). Besides the results on sliding window join, we have also isolated each of the sampling technique and conducted an extensive study on the quality of the sample maintained. This is presented in Appenfix D.

Varying Zipfian

In this experiment, we study the performance of the various window sampling algorithms when the data distribution changes frequently. Two synthetic datasets, D_1 and D_2 are used. Each dataset consists of 250000 tuples. The distribution for each of the dataset changes every 25000 tuples (i.e. Every $0.1|D_1|$ or $0.1|D_2|$). This is achieved by using a zipfian data distribution with Zipfian factor, ζ. For each $0.1|D|$ of data, we randomly generated a ζ factor between 0.0 and 2.0 (inclusive). In addition, to ensure that the skewed values do not cluster within a fixed value range, we also shifted the value ranges for each $0.1|D|$ of data generated.

The results for the experiments are presented in Figure 9.1 to Figure 9.5. Each figure corresponds to a different window size. The window size is expressed as factor of the each of the dataset size. In each of the figures, we present the results for the sliding window join using various sampling technique. We vary the sample size (expressed as a factor of the window size).

In Figure 9.1(a)-(e), the window size is set to $0.02|D|$. We can observe that the *Fifo* and *Res* have high MSE values. This is because *Res* maintains a random sample for the entire data stream, and does not consider the sliding window semantics. Similarly, *Fifo* did not perform as well as it is relatively similar to *Res*. The only difference between *Fifo* and *Res* is that the former does not randomly remove tuples from the sample, but instead removes the earliest tuple from the sample. We note that even though this has an effect of removing older tuple, it is not sufficient to ensure that the sampled data is a good sample for join processing. From the figures, we can also observe that as the sample size increases, the error (i.e. MSE) reduces. Even in the case where the size of the sample is $1.0|W|$, *Fifo* and *Res* still have higher MSE values compared to *Expire* and *WinRes*.

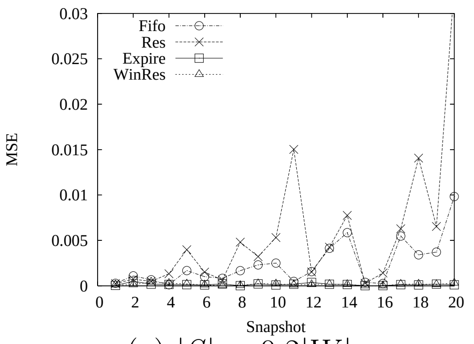
In Figure 9.2(a)-(e), the window size is set to $0.04|D|$. We can observe that both *Fifo* and *Res* have high MSE values, which fluctuates. In contrast, *Expire* and *WinRes* have very low MSE values. This shows that *Expire* and *WinRes* are more effective in maintaining a good result sample, compared with the sliding-window unaware techniques (e.g. *Fifo* and *Res*).

In Figure 9.3(a)-(e), the window size is set to $0.06|D|$. Similarly, we can observe that both *Fifo* and *Res* have high MSE values, compared to *Expire* and *WinRes*. In Figure 9.4(a)-(e), the window size is set to $0.08|D|$. Similarly, we can observe that both *Fifo* and *Res* have high MSE values, compared to *Expire* and *WinRes*.

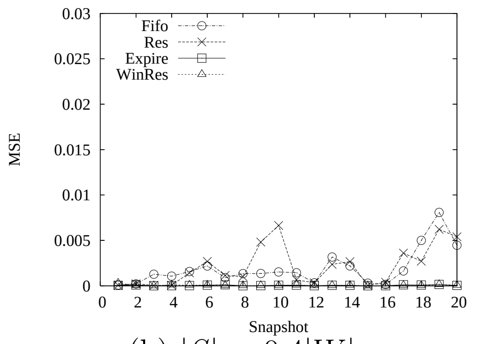
In Figure 9.5(a)-(e), the window size is set to $0.10|D|$. In the experiments, we do not set the window size to be larger than $0.10|D|$. This is because we are interested to study the impact of small window size on sliding window joins. From the figures, we can observe that both *Fifo* and *Res* have high MSE values. In contrast, *Expire* and *WinRes* consistently maintain low MSE. This shows that the approximate results produced by *Expire* and *WinRes* are significantly more accurate compared to *Fifo* and *Res*.

In addition, we also show the zoom of the two window-aware techniques, *Expire* and *WinRes* for varying window size. The graphs are presented in Figure 9.6 to Figure 9.10.

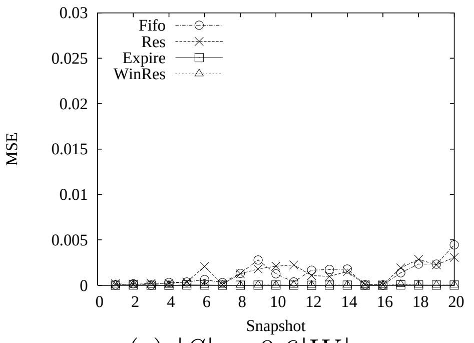
In summary, we can observe from the results that the window-aware techniques, *Expire* and *WinRes* consistently performs much better than the other two window-unaware techniques, *Res* and *Fifo*.



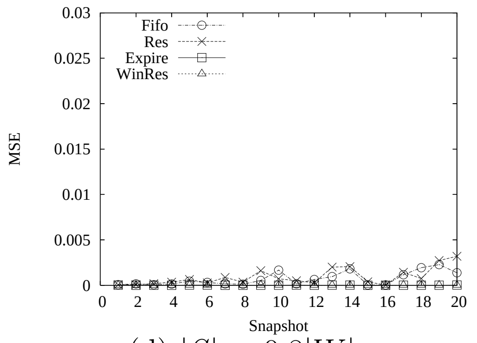
(a) $|S| = 0.2|W|$



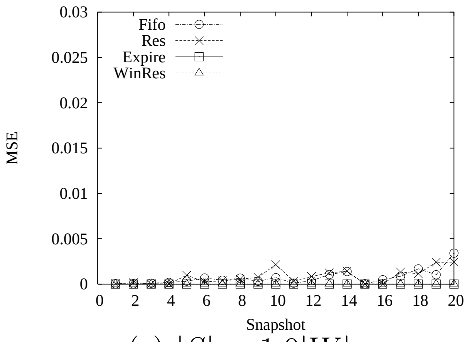
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

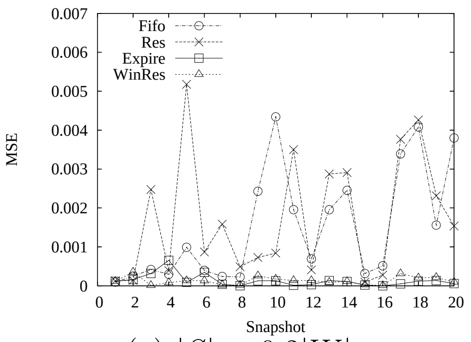


(d) $|S| = 0.8|W|$

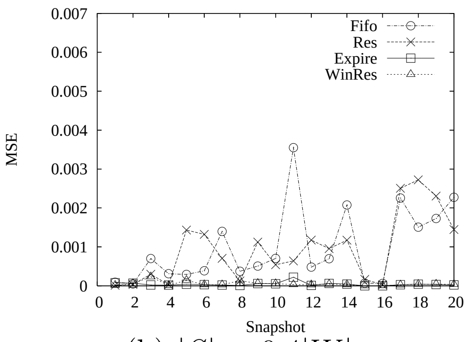


(e) $|S| = 1.0|W|$

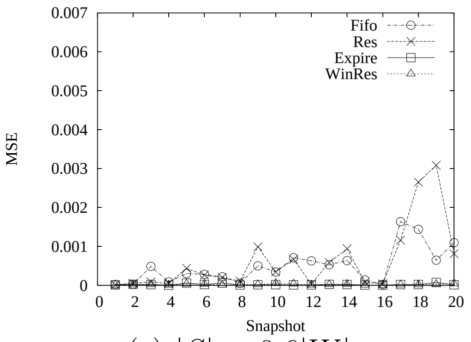
Figure 9.1: Sliding Window Join / Varying Zipfian , $|W| = 0.02|D|$ - MSE vs Snapshots (Note: The maximum MSE is 0.03)



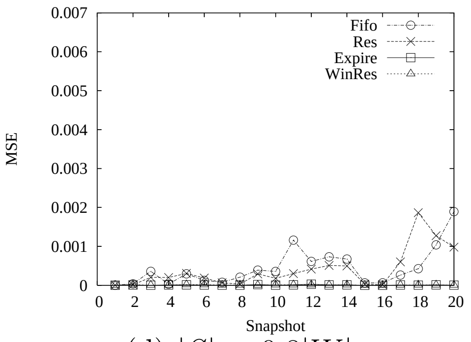
(a) $|S| = 0.2|W|$



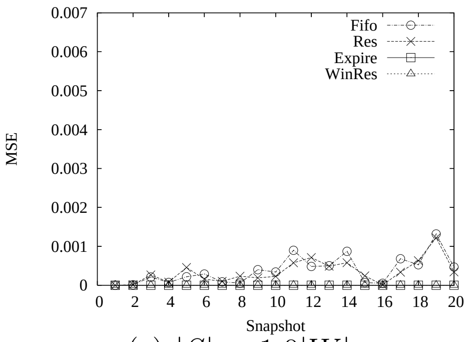
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

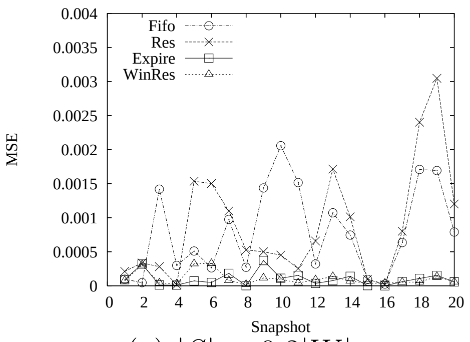


(d) $|S| = 0.8|W|$

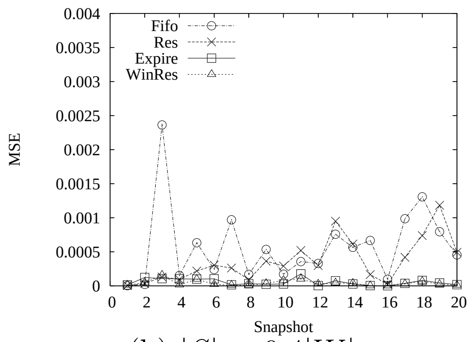


(e) $|S| = 1.0|W|$

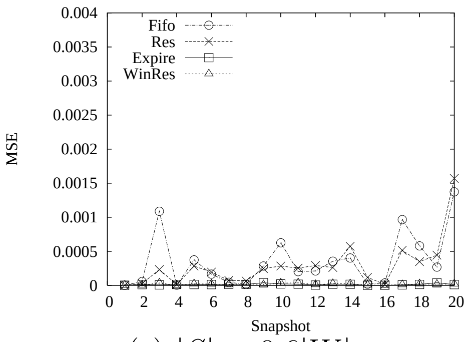
Figure 9.2: Sliding Window Join / Varying Zipfian , $|W| = 0.04|D|$ - MSE vs Snapshots (Note: The maximum MSE is 0.005)



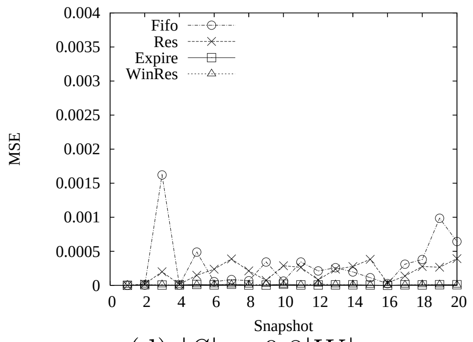
(a) $|S| = 0.2|W|$



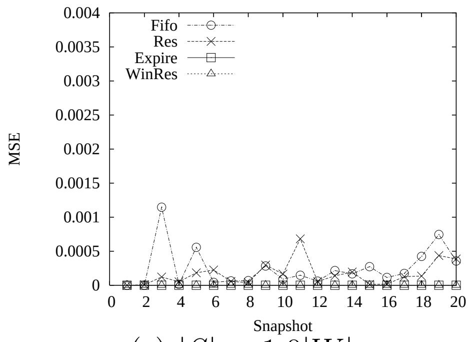
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

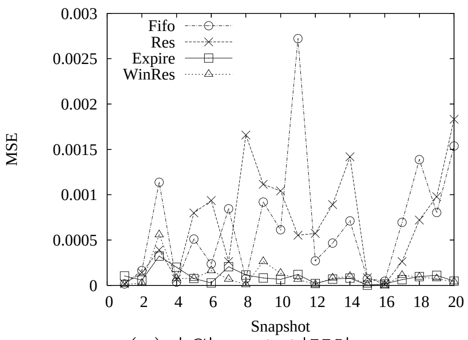


(d) $|S| = 0.8|W|$

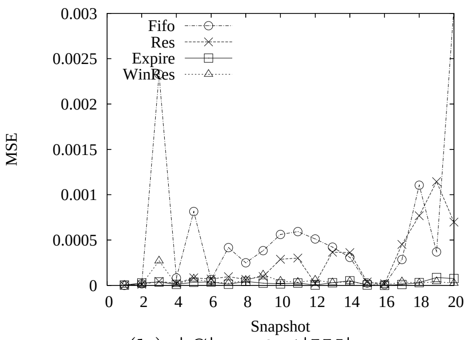


(e) $|S| = 1.0|W|$

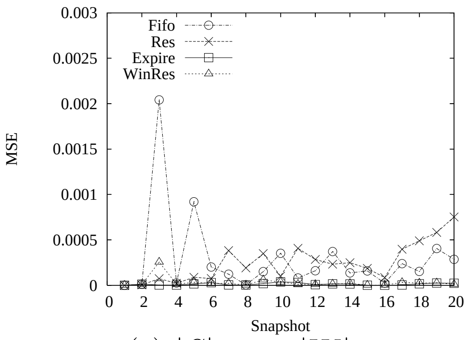
Figure 9.3: Sliding Window Join / Varying Zipfian , $|W| = 0.06|D|$ - MSE vs Snapshots (Note: The maximum MSE is 0.003)



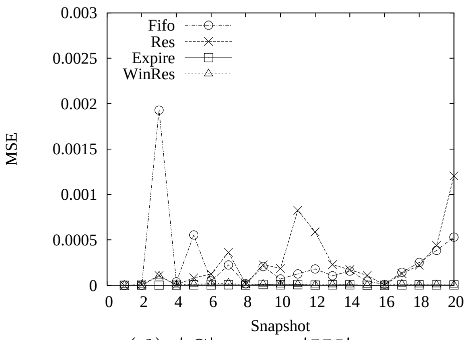
(a) $|S| = 0.2|W|$



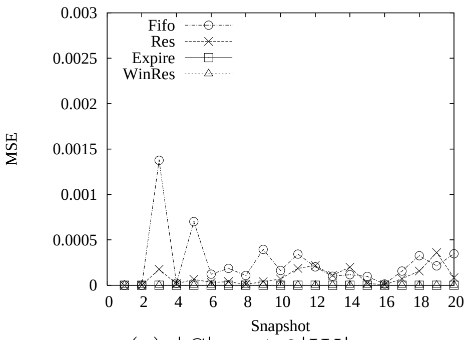
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

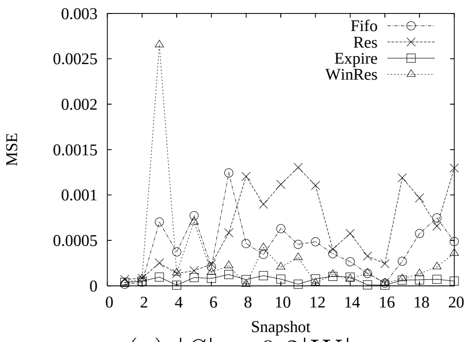


(d) $|S| = 0.8|W|$

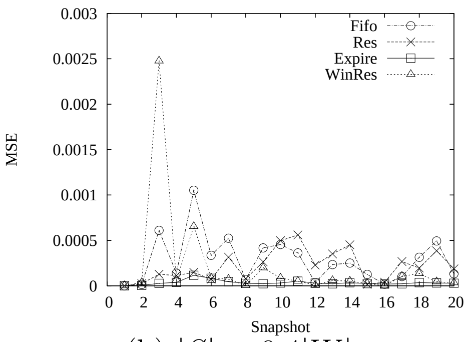


(e) $|S| = 1.0|W|$

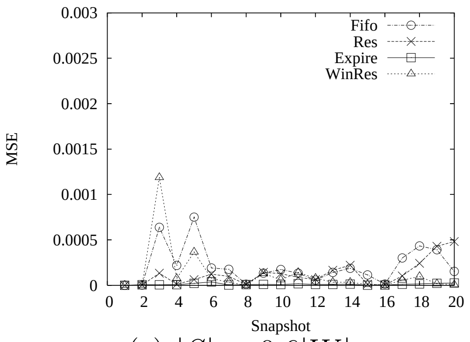
Figure 9.4: Sliding Window Join / Varying Zipfian , $|W| = 0.08|D|$ - MSE vs Snapshots (Note: The maximum MSE is 0.0027)



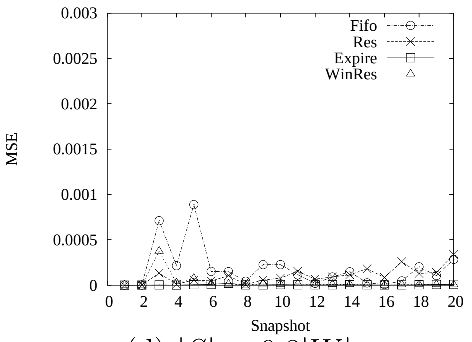
(a) $|S| = 0.2|W|$



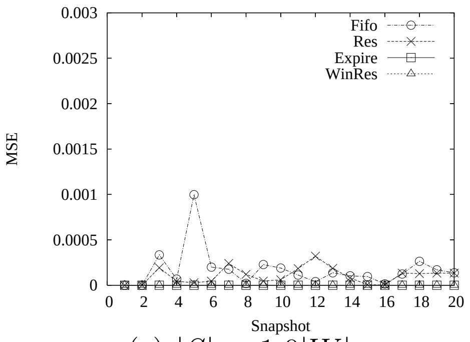
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

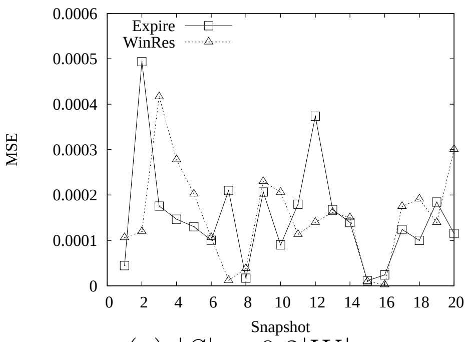


(d) $|S| = 0.8|W|$

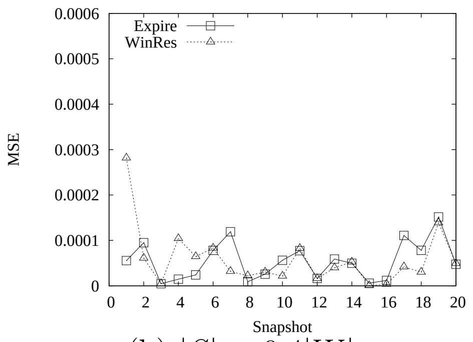


(e) $|S| = 1.0|W|$

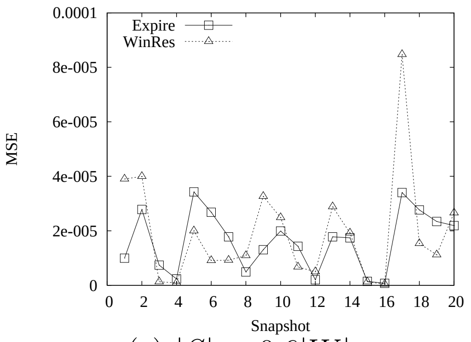
Figure 9.5: Sliding Window Join / Varying Zipfian , $|W| = 0.10|D|$ - MSE vs Snapshots (Note: The maximum MSE is 0.0026)



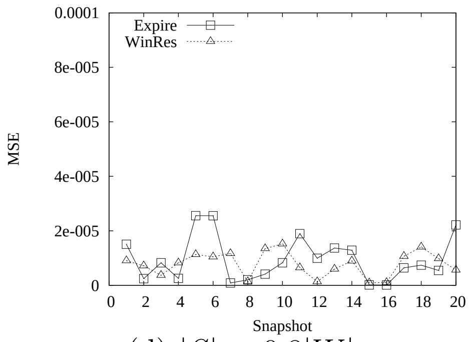
(a) $|S| = 0.2|W|$



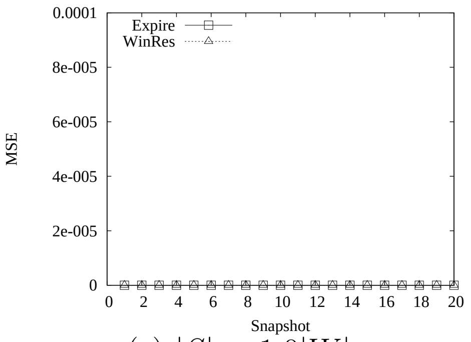
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

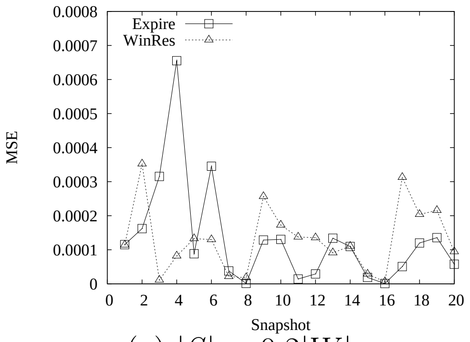


(d) $|S| = 0.8|W|$

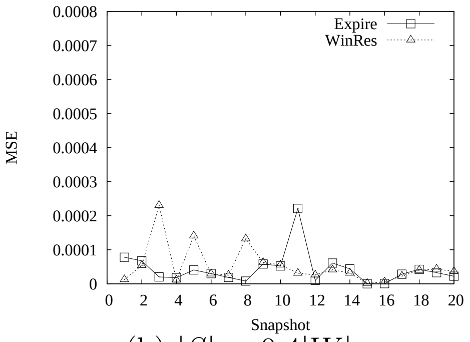


(e) $|S| = 1.0|W|$

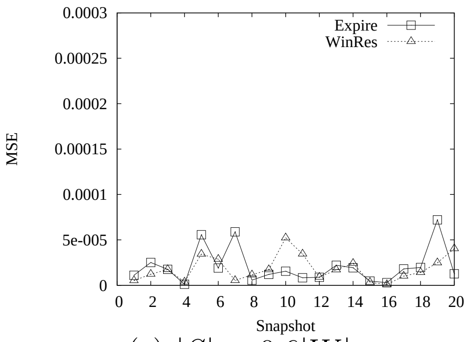
Figure 9.6: (Zoom of Expiry and WinRes) Sliding Window Join / Varying Zipfian , $|W| = 0.02|D|$ - MSE vs Snapshots



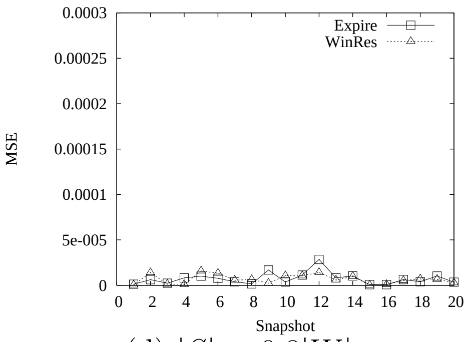
(a) $|S| = 0.2|W|$



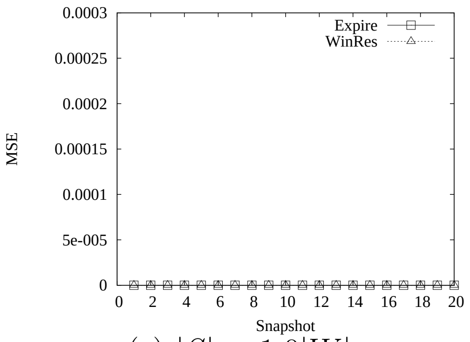
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

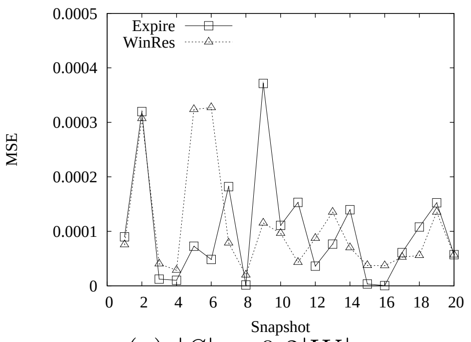


(d) $|S| = 0.8|W|$

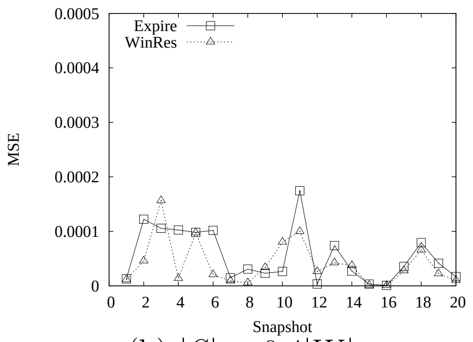


(e) $|S| = 1.0|W|$

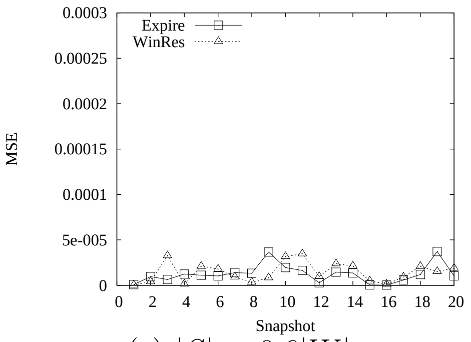
Figure 9.7: (Zoom of Expiry and WinRes) Sliding Window Join / Varying Zipfian , $|W| = 0.04|D|$ - MSE vs Snapshots



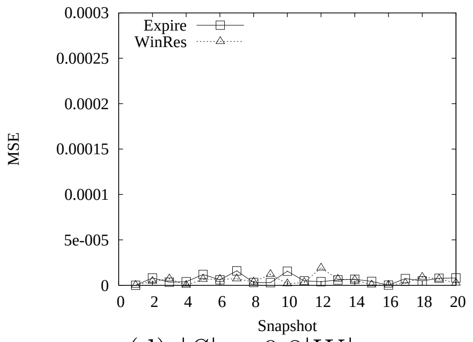
(a) $|S| = 0.2|W|$



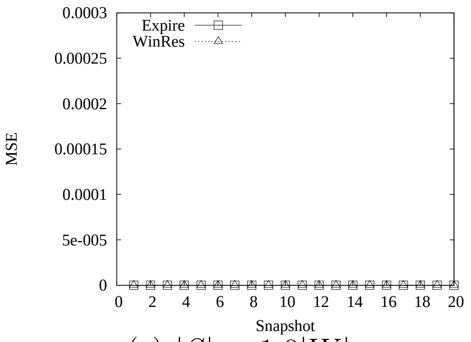
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

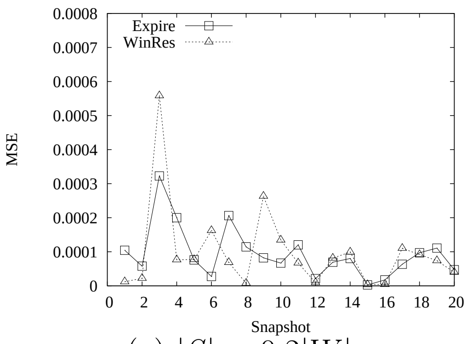


(d) $|S| = 0.8|W|$

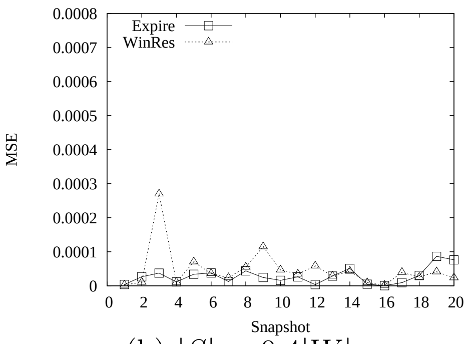


(e) $|S| = 1.0|W|$

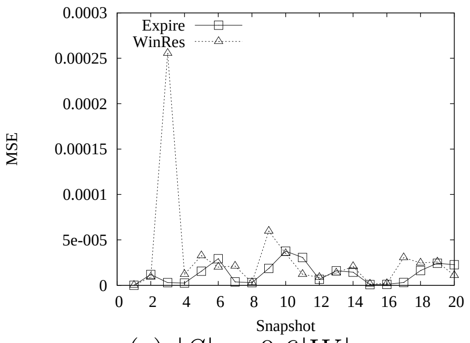
Figure 9.8: (Zoom of Expiry and WinRes) Sliding Window Join / Varying Zipfian , $|W| = 0.06|D|$ - MSE vs Snapshots



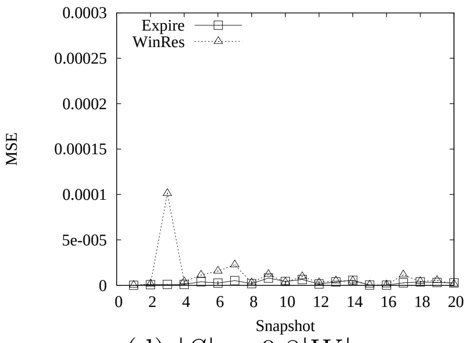
(a) $|S| = 0.2|W|$



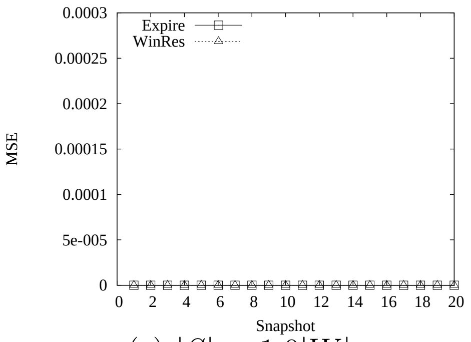
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

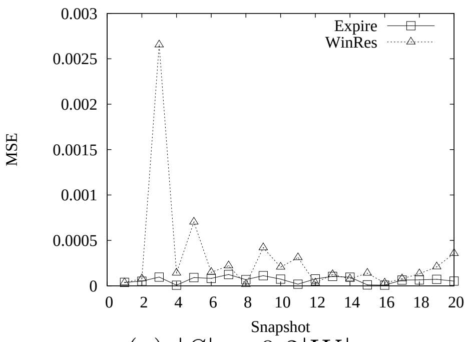


(d) $|S| = 0.8|W|$

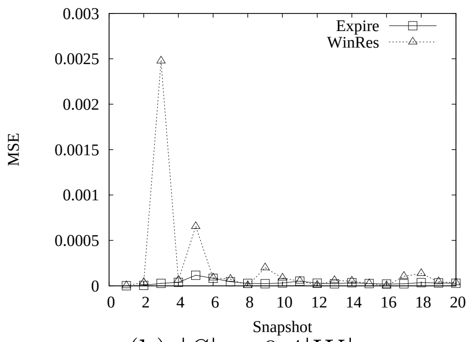


(e) $|S| = 1.0|W|$

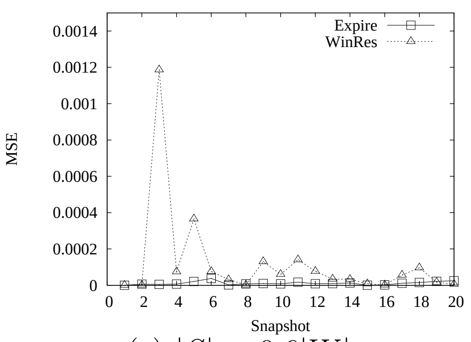
Figure 9.9: (Zoom of Expiry and WinRes) Sliding Window Join / Varying Zipfian , $|W| = 0.08|D|$ - MSE vs Snapshots



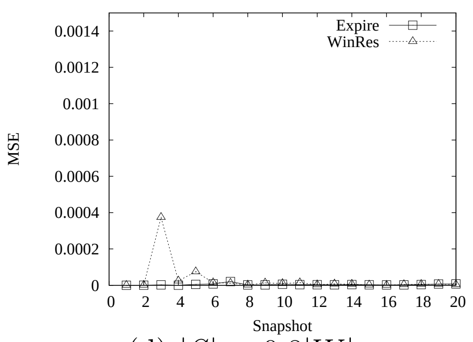
(a) $|S| = 0.2|W|$



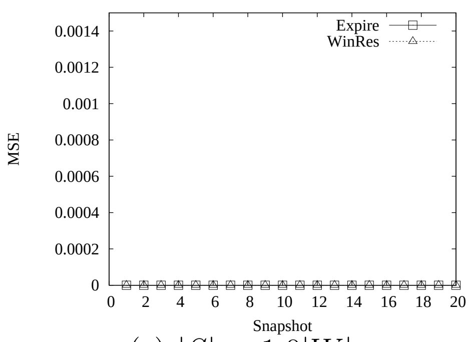
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$



(d) $|S| = 0.8|W|$



(e) $|S| = 1.0|W|$

Figure 9.10: (Zoom of Expiry and WinRes) Sliding Window Join / Varying Zipfian , $|W| = 0.10|D|$ - MSE vs Snapshots

Chapter 10

Conclusion

The universe of network-accessible information is expanding. It is now common practice for applications to process streams of data incoming from remote sources (repositories continuously publishing or sensor networks producing continuous data). In data stream applications, the amount of memory available is limited. Hence, it is important that the memory is effectively used during result production. Amongst the various query processing primitives in data stream applications, the join of data between data streams is an important operation. The design of a progressive join algorithm must meet several key requirements: the algorithm must be non-blocking (or progressive), i.e. it must be able to produce results as soon as possible.The algorithm must maximize either the result quantity or quality.

In this thesis, we developed various techniques to address each of the key requirements. we summarize the main contributions below:

1. Firstly, we studied the design of a progressive join algorithm framework for data stream. We proposed a generic progressive join framework, called Result-Rate based Progressive Join (RRPJ) framework. Using the RRPJ framework, we proposed four instantiations of the framework for different data models models: relational, spatial, high-dimensional and XML. Through extensive performance evaluation, we show that in each of the instantiations, the RRPJ framework is effective in maximizing the number of results produced, and outperforms other state-of-art methods. Most importantly, we show that the RRPJ framework is

generic and can be easily extended to various data models.

To further demonstrate the usefulness of the RRPJ framework, we also developed a system demo, called *Danaides*, for continuous and progressive processing of RSS feeds. The work has been presented in [TBL07a]. *Danaides* uses the RRPJ framework for keeping useful RSS feeds in memory. More details of Danaides can be found in Appendix C.

2. Secondly, we studied the problem of progressive, approximate join. In data stream applications, users often do not require a complete answer to their query but rather only an approximation of the result. They expect the approximation to be either the largest possible or the most representative (or both) given the resources available. In the thesis, we clearly differentiated between the notions of quantity and quality of results produced by progressive approximate join algorithms.

   We proposed four new progressive approximate join algorithms: *ARRPJ*, *ProbHash*, *RAJ* and *RAJHash* are proposed. The former two, like *Prob*, favor quantity, the latter two favor quality. *ProbHash* improves on *Prob* on every aspects. *RAJ* and *RAJHash* produce results of significantly better quality.

3. Thirdly, we studied the problem of progressive, approximate sliding-window join that leverages on sampling as the underlying primitive. We propose several sliding-window sampling techniques which are effective in maintaining a representative sample, and show how they can be used in a sliding window join algorithm.

10.1 Open Issues

In this section, we discuss the list of issues that remain open for further research. We are currently studying the following problems.

In the generic progressive join framework, one of the key factors that contribute to the efficiency and effectiveness of the framework is an effective partitioning method.

A good partitioning method provides a uniform distribution of the data into multiple partitions. This reduces the number of tuples that need to be probed during join processing. In the thesis, we have studied the use of hash partitions for relational and XML data, two-dimension grid for spatial data, and multi-dimension grid for high-dimensional data. However, for both existing and new data models, an open issue lies in finding an effective partitioning scheme. In data stream processing, the data distribution can vary over time. While an effective partitioning scheme can impose a uniform distribution of data into multiple partition for the initial data, it may not be effective for future data that is of a different data distribution. An open issue that needs to be solved is the design of an adaptive partitioning function that can adapt to evolving data.

The work on progressive, approximate joins showed that the use of sampling is an attractive primitive. We are currently studying a unified framework for progressive, approximate join algorithms. The framework focus on balancing between quality and quantity, and allow it to be easily generalized for other data models (e.g spatial, high-dimensional, XML). In this thesis, we have discussed two families of progressive approximate join algorithms which either maximize the quantity or quality of the results produced. *Prob* and *ProbHash* cannot be easily generalized to other data models. This is due to the dependence on the arrival probabilities of the partner data stream. While the arrival probabilities for relational data can be computed in a straightforward manner, it is difficult to compute such probabilities for data from other data models.

Another limitation of *Prob* and *ProbHash* is that they cannot be easily extended for multi-way approximate join, unless the multi-way join query plan is decomposed into a series of binary joins. This is because for a multi-way join, it is not clear which is the partner stream. Decomposing the multi-way join query plan to a series of binary joins would limit the adaptiveness of the join. One of the advantages of using *RAJ* and *RAJHash* is that they can be easily generalized to other data models. We are currently studying the design of multi-way approximate join algorithms based on the *RAJ* and *RAJHash* models. This is because the decision to discard a tuple from the reservoir (or sub-reservoirs) does not depend on the data model. For multi-way joins, multiple reservoirs can be defined for each of the data streams. We are currently investigating the result quality of the answers that are produced using *RAJ* and *RAJHash* for other data models.

In order to address the tradeoff between the two families of algorithms, we are currently looking at tunable sampling. The motivation for tunable sampling is to allow progressive approximate joins to balance between the quantity and quality of results produced. Tunable sampling is defined as a sampling technique which allows users to tune the type of sample produced by the sampling process. The sample can either favor the frequencies for including popular data values in the sample (i.e. quantity), or favor representativeness of the data (i.e. quality). As an initial step, we define a criteria as the parameter to control the type of sample preferred by tunable sampling. Let C denote the set of criterias that the user wishes to maximize, and c_i denotes the individual criteria to be tuned ($c_i \in C$, $1 \leq \mathrm{i} \leq |C|$). Let W denote the set of weights assigned to each criteria, and w_i denotes the individual weight assigned to criteria i. $\sum_{i=1}^{|C|} w_i = 1$. We consider $C = \{$Quantity, Quality $\}$. Next, we introduce the notion of inclusion probability. P(t) is the probability that a tuple t will be included in the sample. We refer to this as the inclusion probability. Given a criteria c_i, the inclusion probability is given by $P_{c_i}(t)$. We formally define tunable sampling as follows: Given a set of criterias C, a set of criteria weights W, and the inclusion probability for each of the criterias. The combined inclusion probability for all the criteria is given by:

$$\sum_{i=1}^{|C|} w_i P_{c_i}(t) \tag{10.1}$$

For quantity maximization techniques (e.g. *Prob*, *ProbHash*), $P_{Quantity} = n_v \; / \; N$, where n_v denotes the number of tuples with value v, and N denotes the total number of tuples that have arrived so far. For quality maximization techniques (e.g. *RAJ*, *RAJHash*), $P_{Quality} = |R| \; / \; N$, where $|R|$ denotes the size of the reservoir, and N denotes the total number of tuples that have arrived so far.

Bibliography

[Agg07] Charu C. Aggarwal. *Data Streams: Models and Algorithms.* Advances in Database Systems. Springer, 2007.

[AGMS99] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, pages 10–20, 1999.

[AKLW07] Mohammed Al-Kateb, Byung Suk Lee, and Xiaoyang Sean Wang. Reservoir sampling over memory-limited stream joins. In *SSDBM*, page 23, 2007.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *ACM symposium on Theory of computing*, pages 20–29, New York, NY, USA, 1996. ACM.

[APR$^+$98] Lars A. Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *VLDB*, pages 570–581, 24–27 1998.

[BBBK00] Christian Böhm, Bernhard Braunmüller, Markus M. Breunig, and Hans-Peter Kriegel. High performance clustering based on the similarity join. In *CIKM*, pages 298–305, 2000.

[BBD$^+$02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[BBKK01] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, pages 379–388, 2001.

[BDL$^+$01] Stéphane Bressan, Gillian Dobbie, Zoé Lacroix, Mong-Li Lee, Ying Guang Li, Ullas Nambiar, and Bimlesh Wadhwa. X007: Applying 007 benchmark to xml query processing tool. In *CIKM*, pages 167–174, 2001.

[BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.

[BGG97] C. Sidney Burrus, Ramesh A. Gopinath, and Haitao Guo. *Introduction to wavelets and wavelet transforms: a primer.* Prentice Hall, 1997.

[BK03] Christian Böhm and Florian Krebs. Supporting kdd applications by the k-nearest neighbor join. In *DEXA*, pages 504–516, 2003.

[BK04] Christian Böhm and Florian Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[CcC$^+$02] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[CDZ06] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An efficient xpath query processor for xml streams. In *ICDE*, page 79, 2006.

[CGRS01] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *VLDB J.*, 10(2-3):199–223, 2001.

[Cla03] J. Clark. The expat xml parser, http://expat.sourceforge.net, 2003.

[CLT$^+$06] Songting Chen, Hua-Gang Li, Jun'ichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over xml documents. In *VLDB*, pages 283–294, 2006.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[CMN98] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD*, 1998.

[CMN99] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274, 1999.

[Coc77] William G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley, 1977.

[DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.

[DGR04] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximation techniques for spatial data. In *SIGMOD Conference*, pages 695–706, 2004.

[DGR05] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Semantic approximation of data stream joins. *IEEE Trans. Knowl. Data Eng.*, 17(1):44–59, 2005.

[DS00] Jens-Peter Dittrich and Bernhard Seeger. Data redundancy and duplicate detection in spatial join processing. In *ICDE*, pages 535–546, 2000.

[DSTW02] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *VLDB*, pages 299–310, 2002.

[EN82] Jarmo Ernvall and Olli Nevalainen. An algorithm for unbiased random sampling. *Comput. J.*, 25(1):45–47, 1982.

[FHK$^+$03] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, Arvind Sundararajan, and Geetika Agrawal. The bea/xqrl streaming xquery processor. In *VLDB*, pages 997–1008, 2003.

[FM83] Philippe Flajolet and G. Nigel Martin. Probabilistic counting. In *FOCS*, pages 76–82, 1983.

[FM85] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[GH05] Sudipto Guha and Boulos Harb. Wavelet synopsis for data streams: minimizing non-euclidean error. In *KDD*, pages 88–97, 2005.

[Gib01] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[GK01] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, pages 58–66, 2001.

[GK02] Sudipto Guha and Nick Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *ICDE*, pages 567–, 2002.

[GKL06] Dieter Gawlick, Muralidhar Krishnaprasad, and Zhen Hua Liu. Using the Oracle database as a declarative RSS hub. In *SIGMOD*, page 722, 2006.

[GKMS03] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. One-pass wavelet decompositions of data streams. *IEEE Trans. Knowl. Data Eng.*, 15(3):541–554, 2003.

[GKS01] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *ACM symposium on Theory of computing*, pages 471–475, New York, NY, USA, 2001. ACM.

[GKS04] Sudipto Guha, Chulyun Kim, and Kyuseok Shim. Xwave: Approximate extended wavelets for streaming data. In *VLDB*, pages 288–299, 2004.

[GKS06] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Trans. Database Syst.*, 31(1):396–438, 2006.

[GLH06] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *VLDB*, pages 595–606, 2006.

[GM98] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, pages 331–342, 1998.

[Gra95] A. Graps. An introduction to wavelets. *Computational Science and Engineering, IEEE*, 2(2):50–61, 1995.

[GSW04] S. Guha, K. Shim, and J. Woo. Rehist: Relative error histogram construction algorithms, 2004.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[HAC$^+$99] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. Interactive data analysis: The control project. *IEEE Computer*, 32(8):51–59, 1999.

[HDG$^+$07] Mingsheng Hong, Alan Demers, Johannes Gehrke, Christoph Koch, Mirek Riedewald, and Walker White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, Beijing, NY, China, 2007. ACM Press.

[HJR97] Y. W. Huang, N. Jing, and E. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *VLDB*, pages 396–405, 1997.

[htta] http://earthquake.usgs.gov/. U.S. geological survey earthquake hazards program.

[httb] http://www.georss.org. GeoRSS:: Geographically encoded objects for rss feeds.

[htt99] http://kdd.ics.uci.edu/. Corel image features dataset, 1999.

[htt06] http://www.microsoft.com/virtualearth/. Microsoft virtual earth, 2006.

[HWL96] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991, http://cdiac.esd.ornl.gov/ftp/ndp026b, 1996.

[Ibr06] Ismail Khalil Ibrahim. *Handbook of Research on Mobile Multimedia (N/A)*. IGI Publishing, Hershey, PA, USA, 2006.

[Ioa03] Yannis E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30, 2003.

[IP95] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD*, pages 233–244, 1995.

[Iva03] Ivelin Ivanov. Processing RSS - http://www.xml.com/pub/a/2003/04/09/xquery.html, 2003.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition.* Addison-Wesley, 1981.

[Koo80] Robert Philip Kooi. *The optimization of queries in relational databases.* PhD thesis, Case Western Reserve University, Cleveland, OH, USA, 1980.

[KP07] Dmitri V. Kalashnikov and Sunil Prabhakar. Fast similarity join for multidimensional data. *Inf. Syst.*, 32(1):160–177, 2007.

[KS97] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *SIGMOD*, pages 324–335, 1997.

[KS98] Nick Koudas and Kenneth C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *ICDE*, pages 466–475, 1998.

[KS00] Nick Koudas and Kenneth C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):3–18, 2000.

[LA05] Xiaogang Li and Gagan Agrawal. Efficient evaluation of xquery over streaming data. In *VLDB*, pages 265–276, 2005.

[Law05] Ramon Lawrence. Early hash join: A configurable algorithm for the efficient and early production of join results. In *VLDB*, pages 841–852, 2005.

[LCKB06] Feifei Li, Ching Chang, George Kollios, and Azer Bestavros. Characterizing and exploiting reference locality in data stream applications. In *ICDE*, page 81, 2006.

[LCL04] Jiaheng Lu, Ting Chen, and Tok Wang Ling. Efficient processing of xml twig patterns with parent child edges: a look-ahead approach. In *CIKM*, pages 533–542, 2004.

[Lin91] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[LLCC05] Jiaheng Lu, Tok Wang Ling, Chee Yong Chan, and Ting Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *VLDB*, pages 193–204, 2005.

[LLZO02] Tao Li, Qi Li, Shenghuo Zhu, and Mitsunori Ogihara. A survey on wavelet applications in data mining. *SIGKDD Explorations*, 4(2):49–68, 2002.

[LMP02] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A transducer-based xml query processor. In *VLDB*, pages 227–238, 2002.

[LNE02] G. Luo, J. F. Naughton, and C. J. Ellmann. A non-blocking parallel spatial join algorithm. In *Intl. Conf. on Data Engineering*, pages 697–705, 2002.

[LR94] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, pages 209–220, 1994.

[LR96] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *SIGMOD*, pages 247–258, 1996.

[MB83] A. I. McLeod and D. R. Bellhouse. A convenient algorithm for drawing a simple random sample. *Applied Statistics*, 32(2):182–184, 1983.

[MLA04] 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 *ICDE*, pages 251–263, 2004.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357, 2002.

[MP99] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for joining multiple inputs. In *SIGMOD*, pages 1–12, 1999.

[MU05] Yossi Matias and Daniel Urieli. Optimal workload-based weighted wavelet synopses. In *ICDT*, pages 368–382, 2005.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD*, pages 448–459, 1998.

[NGSA04] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys*, pages 250–262, 2004.

[NS87] Randal C. Nelson and Hanan Samet. A population analysis for hierarchical data structures. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD*, pages 270–277. ACM Press, 1987.

[OBS99] M. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *Summer Usenix Technical Conference, Monterey*, 1999.

[OKB03] Dan Olteanu, Tobias Kiesling, and François Bry. An evaluation of regular path expressions with qualifiers against xml streams. In *ICDE*, pages 702–704, 2003.

[Olk93] Frank Olken. *Random Sampling from Databases.* Ph.D. dissertation, Computer Science, University of California, 1993.

[OR86] Frank Olken and Doron Rotem. Simple random sampling from relational databases. In *VLDB*, pages 160–169, 1986.

[Par08] Eric Pardede. *Open and Novel Issues in XML Database Applications: Future Directions and Advanced Technologies (To be published).* IGI Publishing, 2008.

[PBF03] Spiros Papadimitriou, Anthony Brockwell, and Christos Faloutsos. Adaptive, hands-off stream mining. In *VLDB*, pages 560–571, 2003.

[PC03] Feng Peng and Sudarshan S. Chawathe. Xpath queries on streaming data. In *SIGMOD*, pages 431–442, 2003.

[pCF99] Kin pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133. IEEE Computer Society, 1999.

[PD96] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[PI97] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, pages 486–495, 1997.

[PIHS96] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD*, pages 294–305, 1996.

[PSC84] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD*, pages 256–276, 1984.

[PWLJ04] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of xquery. In *SIGMOD*, pages 71–82, 2004.

[RG03] R. Ramakrishnan and J. Gehrke. *Database Management Systems, Third Edition*. McGraw-Hill, 2003.

[RSF06] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A complete and efficient algebraic compiler for xquery. In *ICDE*, page 14, 2006.

[rtr] R-tree portal, http://www.rtreeportal.org/datasets.html.

[SFMS07] Michael Stark, Mary Fernández, Philippe Michiels, and Jérôme Siméon. XQuery streaming á la carte. In *ICDE*, 2007.

[SK96] Kenneth C. Sevcik and Nick Koudas. Filter trees for managing spatial data over a range of size granularities. In *VLDB*, pages 16–27, 1996.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. R+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, 1987.

[SSA97] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. High-dimensional similarity joins. In *ICDE*, pages 301–311, 1997.

[SW04] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[SWK$^+$02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[TB02] Wee Hyong Tok and Stéphane Bressan. Efficient and adaptive processing of multiple continuous queries. In *EDBT*, pages 215–232, 2002.

[TBL06] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. Progressive spatial joins. In *SSDBM*, pages 353–358, 2006.

[TBL07a] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. Danaides: Continuous and progressive complex queries on rss feeds. In *DASFAA*, pages 1115–1118, 2007.

[TBL07b] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. Progressive high-dimensional similarity join. In *DEXA*, pages 233–242, 2007.

[TBL07c] Wee Hyong Tok, Stephane Bressan, and Mong-Li Lee. RRPJ : Result-rate based progressive relational join. In *DASFAA*, pages 43–54, 2007.

[TBL08a] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. A stratified approach to progressive approximate joins. In *Proc. of International Conference on Extending Database Technology*, pages 582–593, 2008.

[TBL08b] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. Twig'n join: Progressive query processing of multiple xml streams. In *DASFAA*, pages 546–553, 2008.

[TGIK02] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. Dynamic multidimensional histograms. In *SIGMOD*, pages 428–439, 2002.

[TVB$^+$02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, 2002.

[TYP$^+$05] 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*, pages 371–382, 2005.

[UF99] Tolga Urhan and Michael J. Franklin. XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994, University of Maryland, 1999.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD*, pages 130–141. ACM Press, 1998.

[Vit84] Jeffrey Scott Vitter. Faster methods for random sampling. *Commun. ACM*, 27(7):703–718, 1984.

[Vit85] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[VNB03] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[VW99] Jeffrey Scott Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD*, pages 193–204, 1999.

[WA91] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *PDIS*, pages 68–77, 1991.

[XLOH04] Chenyi Xia, Hongjun Lu, Beng Chin Ooi, and Jin Hu. Gorder: An efficient method for knn join processing. In *VLDB*, pages 756–767, 2004.

[XML02] XML Data Repository. http://www.cs.washington.edu/research/xmldatasets/, 2002.

[XYC05] Junyi Xie, Jun Yang, and Yuguo Chen. On joining and caching stochastic streams. In *SIGMOD*, pages 359–370, 2005.

[Yah07] Yahoo Pipes. http://pipes.yahoo.com/pipes/, 2007.

Appendix A

Initial Study on Progressive Spatial Join

Algorithms for the processing of spatial join, such as those proposed in [BKS93, LR94, LR96, PD96, APR$^+$98], assume that the data is organized and readily available on local disks. These algorithms emphasize the efficient processing of the complete result of the spatial join. We refer to these algorithms as *blocking spatial join* algorithms for they require both data sets to be available and possibly indexed before results are requested.

The modern information infrastructure is one of networked devices possibly mobile and wireless. It enables the production and consumption of huge amounts of data. The applications feeding on these data either need to process continuous streams of spatial data or require the processing of quantities of spatial data so huge that they render the existing blocking algorithms impractical for a user waiting for results. They compel spatial join algorithms that can swiftly deliver initial results with the minimum negative impact on the overall response time, i.e. *non-blocking spatial join* algorithms.

The first family of parallel non-blocking spatial join algorithms is proposed and studied in [LNE02]. While the focus of the work is on achieving speed-up by distributing the task of performing the spatial join amongst several processors, the authors considered non-blocking spatial join with a transient in-memory R-tree index structure.

A.1 R-tree Based Blocking and Non-Blocking Spatial Joins

A.1.1 Static Spatial Join

Let us first recall the general strategy of an R-tree based blocking spatial join. For the sake of using the performance of this algorithm as a base line in the subsequent performance analysis, we can consider, without loss of generality for the non-blocking algorithms that we propose, that the data sets are bounded. The blocking R-tree based spatial join first builds two R-trees one for each incoming data set. We do not use bulk loading which would further delay the production of results. When all the data has arrived, a synchronized traversal of the R-tree is used to compute overlapping data (since we are concerned only with the filtering phase, data consist of an identifier to refer to the actual spatial object and the four coordinates of a minimum bounding rectangle.) This strategy is the basis of algorithms such as those in [BKS93, LR94, LR96, PD96, APR$^+$98] even though details of the underlying data structure and algorithms might differ. Since we consistently use R-trees, we believe that the relative performance is generally similar to the one we would obtain with more sophisticated index structures such as R+-trees, R*-trees, and their variants.

Figure Algorithm 14 outlines the algorithm for joining the two data sets R and S by constructing the two R-trees P_R and P_S to guide join processing.

We can identify two distinct phases in this generic framework. In the *Build* phase the index is build. In the *Join* phase the indices are use to guide the production of results. If extending this algorithm, non-blocking algorithms need to interleave the build and join phases in order to allow the early production of results.

Algorithm 14 Static Spatial Join Algorithm

```
1: Build Phase
2: $P_R$ and $P_S$ are intermediate data structures
3: for Tuple t ∈ R do
4:     Insert t into $P_R$
5: end for
6: for Tuple t ∈ S do
7:     Insert t into $P_S$
8: end for
9: Join Phase
10: Join($P_R$,$P_S$)
```

A.1.2 Fully Dynamic Spatial Join

The first non-blocking algorithm that comes to mind, considering the above discussed blocking algorithm, consists in the interleaving of the two phases at finest granularity. A system-based concurrent execution of the two phases that would rely on concurrency control of the R-tree accesses is not necessary since each insertion preemptively locks the root of the tree to allow potential splits to retro-propagate up to the root if necessary. It suffices to programmatically alternate the two phases. Namely each incoming data from either set is inserted into its corresponding R-tree and used to probe the other data sets already partially build R-tree. We call this algorithm the *fully dynamic spatial join.*The fully dynamic spatial join algorithm is outlined on Algorithm 15.

Algorithm 15 Fully Dynamic Spatial Join Algorithm

```
1: Given Spatial Relations R and S
2: while (Data Available) do
3:     Read a tuple from either of the data $R$ or $S$
4:     Insert tuple into R-tree (for R or S)
5:     Probe other R-tree using tuple
6:     Return MBRs which overlap
7: end while
```

Clearly, this algorithm will produce the first results very early. Yet we can expect its overall performance to be much worse than the one of the blocking spatial join algorithm. This is noted in [LR94]: *If we simply used R-tree and let them overflow to* *disk when they grow larger than main memory, performance would not be acceptable.*. Indeed one of the dominant costs, namely the amount of retrieval of data pages from disk, is commensurate to the amount of probing (ultimately the sum of the size of both data sets.) Although this cost is reduced by the use of a buffer and by an adequate replacement policy such as the *least recently used* or *LRU* policy, it can only be done within the limit of the available space available for the buffer (a fortiori so if data sets are unbound).

A.1.3 Block Fully Dynamic Spatial Join

In order to seek a compromise between the minimum number of input-output operations required by the blocking algorithm and the non-blocking behavior of the fully dynamic algorithm, we propose to alternate the insert and join phases for blocks of data. Namely whenever we have received a predefined number of data to form a block from one of either set, the block of data is inserted into its corresponding R-tree and the disjunctive list of data is used to probe the other data sets already partially build R-tree. We call this algorithm the *block fully dynamic spatial join* algorithm.

The algorithm is outline on Algorithm 16.

Algorithm 16 Block Fully Dynamic Spatial Join Algorithm

```
1: Spatial Relations R and S, BlkThreshold T
2: while (Data Available) do
3:    Receive a $tuple_i$ from either R or S
4:    Insert $tuple_i$ into $TupleCollection_i$
5:    if (size($TupleCollection_i$) >= T) then
6:       Insert all tuples in $C_i$ into corresponding R-tree
7:       Use all tuples in $C_i$ to probe the corresponding R-tree
8:       Return overlap MBRs (if found)
9:       Empty $TupleCollection_i$
10:   end if
11: end while
```

The size of the blocks determines the compromise between the early production of results (small blocks) and the overall performance (large blocks). For reasons of symmetry (assuming identical arrival frequency on both data sets) a size of half of the buffer yields the optimum overall performance.

A.1.4 R-tree Based Non-Blocking Spatial Joins

The R-tree based non-blocking spatial joins should yield interesting performance in the production of early results while not compromising the performance of the overall production of results as long as the input-output time saved by retrieving relevant pages thanks to the R-tree and the cpu time saved by comparing spatially related data overcomes the cost of the repeated join phase. The questions are whether this cross-over occurs after a sufficient percentage of the data has been produced and how much overhead is incurred at completion of the join (for finite data sets).

In other attempts, whose full details are not reported here, we have considered variants of the non-blocking algorithms described above in which the partially build R-trees are joined instead of being probed with a list of data as well as strategies for inserting data as they arrive and for marking them to avoid duplicate results. The empirical analysis showed poor performance compared to the algorithms discussed in this thesis.

A.1.5 Symmetric Block Nested Loop Algorithm

The main purpose of the R-tree is to adaptively create a balanced partition of the data. Other partitioning technique such as grids or quad-trees either degenerate if the data is skewed in a way not captured by the partition or introduce may introduce similar overhead to the one of the R-tree for a similar granularity of partitioning. Given the expected prohibitive cost of managing a disk resident R-tree, we can consider an even more radical solution, namely an algorithm that solely focuses on reducing the input-output operations with respect to the buffer without attempting to partition the data. In conventional relational database management systems, if no relevant index data structure exists on either of the data sets to be joined, one of the most common join algorithms is the Block Nested Loop Join [RG03]. In this section we propose a *symmetric block nested loop* algorithm. As a matter of fact such an algorithm applies equally to spatial and non-spatial data since no particular organization of the data is needed which depends on its spatial nature. In the relational context with adequate join conditions on pairs of attributes one can consider efficient dynamic partitioning functions such as hash functions (yielding algorithms such as the Xjoin [UF99], for instance). Such partitioning functions so far have found no equivalent in the spatial domain, and are not readily available for arbitrary join conditions in general in other domains.

Algorithm

The fact that we are dealing with pages instead of data elements allows us a tighter control of the buffer. In the symmetric block nested loop algorithm we partition the buffer of size B into three groups. We allocate two buffers of $n = (B - 1)/2$ frames (to hold one block of n pages) to read in data from each of the two data sets. Two counters are kept to indicate when a full block of data is read from either data set. When full, the block of data is joined in a nested loop with the already disk resident data of the other data set. In addition, a single buffer frame is reserved to read from the disk the data to be joined. The build phase is reduced to reading and storing the data since no index is built. The pages in the each block are written to disk as new data is read according to the LRU replacement policy. This occurs after the data in the buffer have been joined thus not necessitating duplicate elimination in the results. Algorithm 17 outlines this algorithm.

Algorithm 17 Symmetric Block Nested Loop Algorithm

```
1: Spatial Relations R and S, BlkThreshold T
2: while (Data Available) do
3:    Read a tuple from either of the data $source_i$
4:    Insert tuple into buffer $B_i$
5:    $\text{BlkCounter}_i$++
6:    if ($\text{BlkCounter}_i$ >= T) then
7:       for (each stored page of the other data set) do
8:          Join this page with the data in $B_i$
9:       end for$\text{BlkCounter}_i = 0$
10:   end if
11: end while
```

An additional noticeable advantage of this algorithm is that data is written in pages in its order of arrival as opposed to being reorganized as in the algorithms given in the previous section. Provided pages or data are time-stamped, this feature simplifies the task of discarding outdated data if the application requires it.

If the data displays no particular pattern of arrival with respect to its spatial distribution and in the impossibility to find a satisfactory and economic partitioning mechanism, we expect the symmetric block nested loop algorithm to be competitive.

A.1.6 Using R-tree for Dynamic Spatial Join

We now consider an algorithm suitable for those applications in which data is expected to arrive in spatial clusters. In such a case, except at the transition between two arriving clusters, we can expect a sequence of incoming data from one data set, say of the size of one page, to be spatially near.

Summary R-tree

Based on the above assumption the algorithm we propose uses an R-tree to index pages instead of individual data (notice that the approach naturally extend to considering groups of several pages if the data sets are very large and the clustering sufficient). For each page of data read from each data set, the minimum bounding rectangle in closing the data in the page is stored in the R-tree for this data set. We call such R-trees *summary R-trees.* Notice that this is different from bulk loading the actual data in the page since we do not index the actual data but the page that contains them. The size of the summary R-tree is much smaller than the one of an R-tree indexing the actual data. Figures A.1 and A.2 illustrate the layout of the data in the directory and leaf pages in a complete R-tree and in a corresponding summary R-tree, respectively, for the R100C5 dataset (see section A.1.7), which contains 100K of data and has five clusters. We see that the summary R-tree still contains the five clusters although it is one level shorter.

Symmetric Indexed Block Nested Loop

This strategy suggests a new algorithm we call the *symmetric indexed block nested loop.* The Symmetric block nested loop follows the block fully dynamic join algorithm

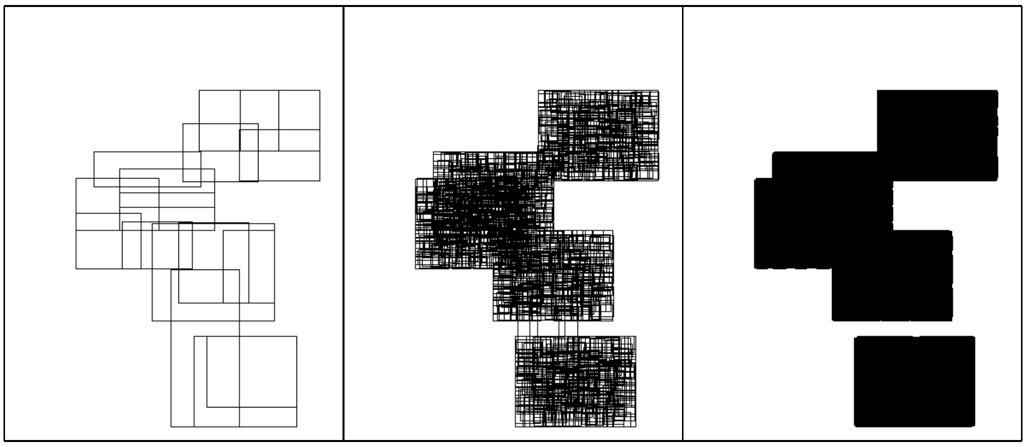


Figure A.1: R-tree layout for R100C5

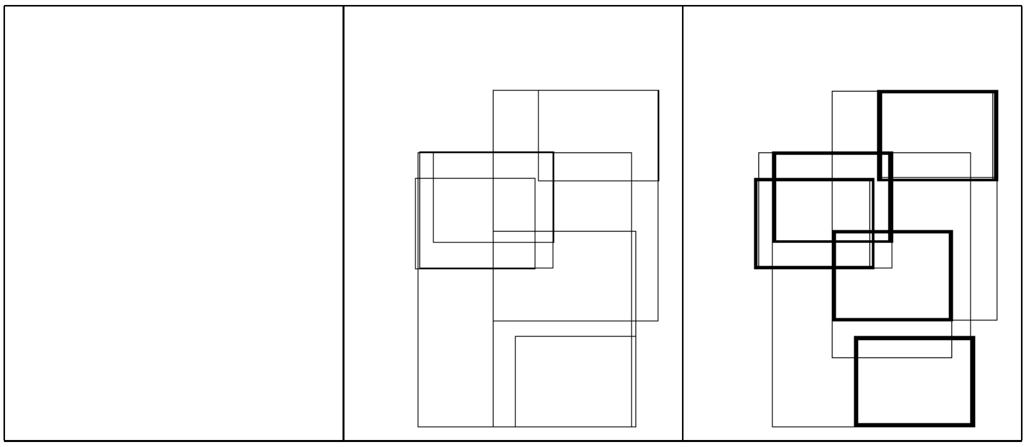


Figure A.2: Summary R-tree layout for R100C5

of section A.1.3. The fact that we are dealing with pages instead of data elements, as in the case of the block nested loop algorithm, allows us a tighter control of the buffer.

In the symmetric indexed block nested loop algorithm we partition the buffer of size B used in the above algorithm into five groups. We allocate three frames to each R-tree. We allocate two buffers of $n = (B - 7)/2$ frames to read in data from each of the two data sets. Two counters are kept to indicate when a block of full pages of data is read from either data set. The size of the block is n.

When a block of full pages of data is read, the minimum bounding rectangles of each page in the block is inserted into the corresponding R-tree. The disjunctive list of minimum bounding boxes is used to probe the other already partially build R-tree. The data in the pages retrieved are joined with the data in the pages in the block.

The pages in each block are written to disk as new data is read according to the LRU replacement policy. This occurs after the data in the buffer have been joined thus not necessitating duplicate elimination in the results. The algorithm is presented in Algorithm 18.

Algorithm 18 Symmetric Indexed Block Nested Loop Algorithm

```
 1: Spatial Relations R and S, BlkThreshold T
 2: while (Data Available) do
 3:    Receive a $tuple_i$ from either of the data $source_i$
 4:    Insert $tuple_i$ into bucket $B_i$
 5:    BlkCounter$_i$++
 6:    if (BlkCounter$_i$ >= T) then
 7:       MBRList = List of Covering MBRs
 8:       FoundList = Use MBRList as query windows in the
 9:       summary R-tree of the other data source
10:       Perform Block-Nested Loop Spatial Join $B_i$ with
11:       with pages in FoundList
12:       BlkCounter$_i$ = 0
13:    end if
14: end while
```

Reflecting the natural clustering of the data, the summary R-tree reduces the number of pairs of pages to be selected for joining the data they contain. The question is whether and at which level of clustering this savings overcome the cost of creating and maintaining the summary R-tree.

This algorithm also maintains the advantage that data is written in pages in its order of arrival as opposed to being reorganized as in the algorithms given in the previous section. Provided pages or data are time-stamped, this feature simplifies the task of discarding outdated data if the application requires it although entries in the summary R-tree might need to be discarded or might become obsolete.

A.1.7 Performance Analysis

Experimental Set-up

The algorithms are implemented in C. The experiments run on a Pentium 4 1.6GHz PC with 512MB RAM under Windows XP Professional. The input-output operations simulate a state of the art disk spinning at 7200rpms yielding an input-output cost of 8ms. We use a 128 frames buffer for all the algorithms. One frame holds one page. The size of a page is 4096 bytes.

We use both synthetic and real-life datasets. Without loss of generality, we do consider data sets that contain an identifier to the actual spatial object as well as its minimum bounding rectangle. One data record is of size 20 bytes. Unless stated otherwise, inter-arrival rate is constant.

The synthetic data sets are generated using a generator similar to the one described in [LR94]. The generation allows us to control the number of clusters of the original data distribution as well as the selectivity of the join. Two datasets of size N are generated as follows: We first randomly generate C clusters centers for the first data set. For each cluster center, we generate cluster rectangles, CR. Both the length and width of each cluster rectangle is set at 0.2 in our experiments. We assigned $\lfloor N/C \rfloor$ data rectangles to each cluster. The remaining data rectangles are then randomly assigned to any cluster. To control the selectivity S, the second data set clusters are constructed such that $S\%$ of their area overlaps with a cluster from the other dataset. Data from the same cluster are contiguous. For some experiments, when indicated, the data is randomly reshuffled.

The realistic data sets are the Greek roads and rivers [rtr] and the German roads and railroad lines [rtr]. A summary of characteristics of these data sets is presented in Table A.1.

R-tree Based Blocking and Non-Blocking Spatial Joins

In this experiment, we analyze the performance of the three R-tree based algorithms.

We first use two synthetic data sets of 100K each, with 5 clusters each, and with a join selectivity of 25%. We compare the performance of the static spatial join, the

Datasets	Number of MBRs	# of Clusters
Real-life		
Greece		
Rivers	24,650	-
Roads	23,268	-
Germany		
Railroad Lines	36,334	-
Roads	30,674	-
Synthetic		
R50KC5, S50KC5	50,000	5
R100KC5, S100KC5	100,000	5
R100KC10, S100KC10	100,000	10
R100KC20, S100KC20	100,000	20
R200KC5, S200KC5	200,000	5
R400KC5, S400KC5	400,000	5

Table A.1: Datasets Used

fully dynamic spatial join, and the block fully dynamic spatial join.

Figures A.3(a), A.3(b) and A.3(c) report the cumulated number of input-output operations of each of the three algorithms, respectively, at varying percentage of results produced. On the figure the input-output operations occurring during the build phase (insertion of the data and creation of the R-tree) are in grey, while the input-output operations occurring in the join phase are in black. Notice that the figures have different scales on the y-axis.

Figure A.3(d) reports the cumulated response time of each of the three algorithms at varying percentage of results produced.

The response time charts confirms that both the fully dynamic and the block fully dynamic joins can produce results early. At what cost for their overall performance?

By design, the build phase of the static spatial join occurs before any results can be produced. Both the fully dynamic spatial join and the block fully dynamic spatial join successfully distribute the build phase and its input-output operations during the incremental production of results. For the fully dynamic spatial join, the input-output cost of joining each individual data is prohibitive. For the block dynamic join the input-output cost remains similar to the one of the static spatial join.

This respectable performance input-output of the block fully dynamic cannot be maintained for the overall response time. Both dynamic algorithms incur a prohibitive cpu cost. Indeed, while the static algorithm is joining the two R-trees in a single depth-first traversal (see [BKS93], for instance), both dynamic algorithms probe the R-trees for each individual or list of minimum bounding rectangles. Their overall performance in response time is worse than the one of the static spatial join. The fully dynamic algorithm can produce more than 20% of the results faster than the static algorithm on this data set. The block fully dynamic algorithm can produce more than 60% of the results faster than the static algorithm on this data set.

Symmetric Block-Nested Loop

In this experiment, we compare the performance of the symmetric block nested loop (SBNL) algorithm with those of the static spatial join algorithm. We use two synthetic data sets of 100K each, with 5 clusters each, and with a join selectivity of 25%.

Figure A.4(a) reports the cumulated number of pages actually compared during the execution of each of the three algorithms, respectively, at varying percentage of results produced. Figure A.4(b) reports the cumulated number of input-output operations of each of the three algorithms, respectively, at varying percentage of results produced. Figure A.4(c) reports the cumulated response time of each of the three algorithms, respectively, at varying percentage of results produced.

We see that although many more pages are compared by the block nested loop (each pair of pages, one from each data set, is ultimately compared by this algorithm), this is translated in a reasonably low number of input-output operations thanks to the absence of the index data structure to build and probe and thanks to the buffer. Not only the block nested loop can create early results faster than the static algorithm, but it creates all the results significantly faster than the static algorithm. We consistently observed this pattern of performance for all the data sets we have tried (see subsection A.1.7).

Symmetric Indexed Block Nested Loop and Clustered Arrivals

In this experiment, we analyze the effect of clustered arrivals on the symmetric block nested loop and the symmetric indexed block nested loop (SIBNL) algorithms. We use two synthetic data sets of 100K each, with 5 clusters each, and with a join selectivity of 25%. In a first series of measurements the data is clustered as generated, while in a second series of measurements the data from both data sets is randomly reshuffled.

We compare the performance of the static spatial join, the symmetric block nested loop, and the symmetric indexed block nested loop for both pairs of data sets.

For the clustered data set, the results are reported on figures A.5(a), A.5(b), and A.5(c) as mentioned in the previous subsection.

We first observed that, as motivated by the design of the symmetric indexed block nested loop, it can reduce the number of pages being compared. This means that it does filter relevant pages of data. Yet because of the cost of maintaining and probing the index data structure, although just a summary, this performance does not translate into a commensurate gain in input-output cost and response time. Nevertheless, with these data sets arriving in clusters, the symmetric indexed block nested loop manages to yield a better response time than the symmetric block nested loop.

For the randomly shuffled data set, Figure A.5(d) reports the cumulated number of pages actually compared during the execution of each of the three algorithms, respectively, at varying percentage of results produced. Figure A.5(e) reports the cumulated number of input-output operations of each of the three algorithms, respectively, at varying percentage of results produced. Figure A.5(f) reports the cumulated response time of each of the three algorithms, respectively, at varying percentage of results produced.

The relative performance of the symmetric indexed block nested loop and the symmetric block nested loop as shown on figures A.5(b), and A.5(c) is now reversed on figures A.5(e), and A.5(f). This illustrates that the symmetric indexed block nested loop can exploit situations in which data arrive in clusters yet it still performs reasonably well when this is not the case.

The results are accentuated when the data distribution is more clustered and the arrival clustered for instance with 10 and 20 clusters (using R100KC10 $\bowtie$ S100KC10 and R100KC20 $\bowtie$ S100KC20).

Scalability and Real-Life Data Sets

In this series of experiments, we measure the performance in both input-output operations and response time for the symmetric block nested loop and the indexed block nested loop algorithms on both very large and real-life data sets, respectively. For reference, we also measure the performance of the static spatial join.

We use two groups of synthetic data sets of 200K and 400K. All data sets have 5 clusters and the selectivity is 25%.

For the two input data sets of size 200K, Figure A.6(a) reports the cumulated number of input-output operations of each of the three algorithms, respectively, at varying percentage of results produced. Figure A.6(b) reports the cumulated response time of each of the three algorithms at varying percentage of results produced.

We use two very large synthetic data sets of 400K,

For the two input data sets of size 400K, Figure A.6(c) reports the cumulated number of input-output operations of each of the three algorithms, respectively, at varying percentage of results produced. Figure A.6(d) reports the cumulated response time of each of the three algorithms at varying percentage of results produced.

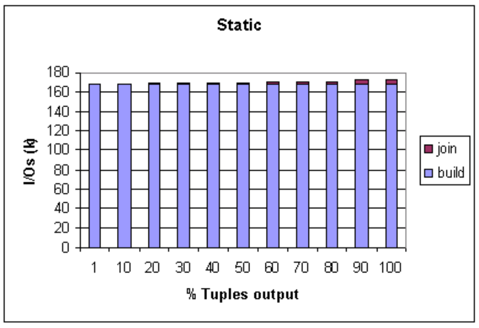
We use the Greek road and rivers and the German roads and railroads data sets. Figure A.7(a) reports the cumulated number of input-output operations of each of the three algorithms, respectively, at varying percentage of results produced for the Greek data. Figure A.7(b) reports the cumulated response time of each of the three algorithms at varying percentage of results produced for the Greece data. Figure A.7(c) reports the cumulated number of input-output operations of each of the three algorithms, respectively, at varying percentage of results produced for the German data. Figure A.7(d) reports the cumulated response time of each of the three algorithms at varying percentage of results produced for the Germany data.

These charts call no further analysis since they are only presented to confirm the analysis in the previous subsections with more challenging data sets.

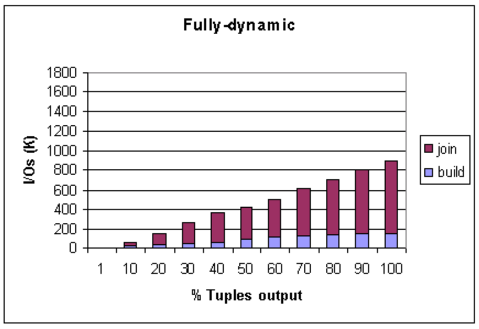
Inter-arrival Rate

In this experiment, we illustrate the effect of non-constant inter-arrival times on the various algorithms. We use two synthetic data sets of 50K each, with 20 clusters each, and with a join selectivity of 25%. The inter-arrival is modeled using a Poisson law with a mean of 2 seconds. We compare the performance of the static spatial join, block dynamic spatial join, the symmetric block nested loop, and the symmetric indexed block nested loop.

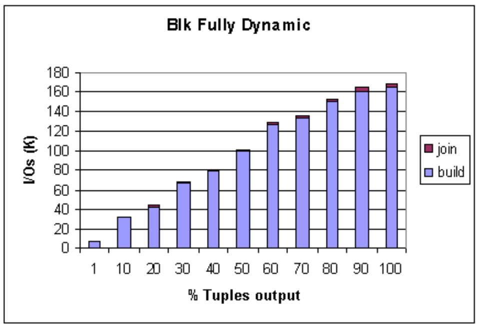
Figure A.8 reports the cumulated response time of each of the four algorithms at varying percentage of results produced for the Poisson law. We can observe that the symmetric block nested loop and symmetric indexed block nested loop were able to produce the initial 75% of the results quickly in spite of the inter-arrival rate.



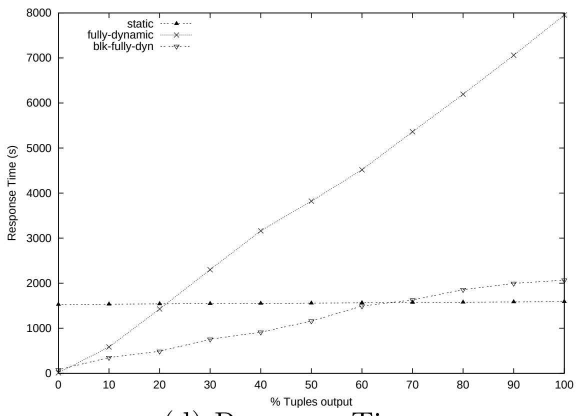
(a) Static



(b) Fully Dynamic



(c) Block Fully Dynamic



(d) Response Time

Figure A.3: Comparison of R-tree Based Spatial Joins (R100KC5 ⋈ S100KC5)

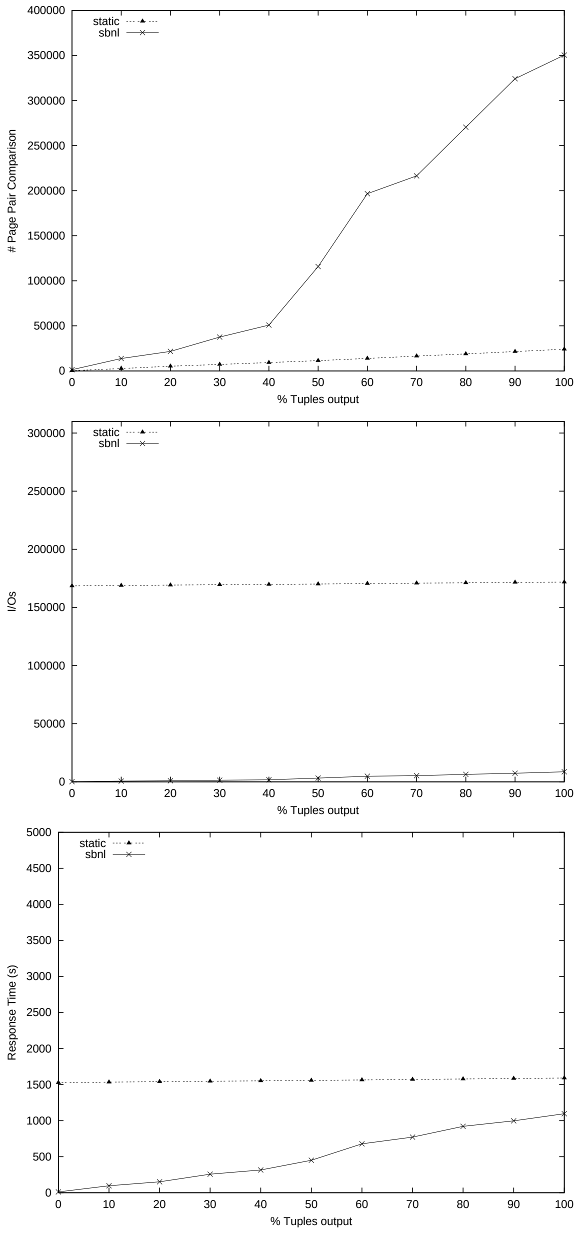
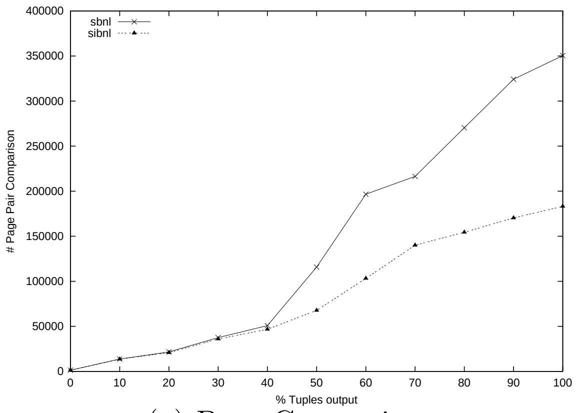
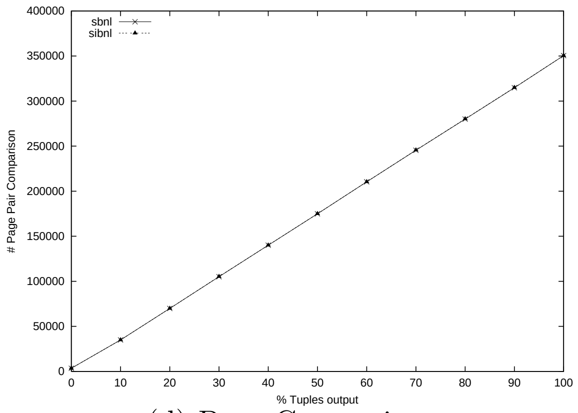


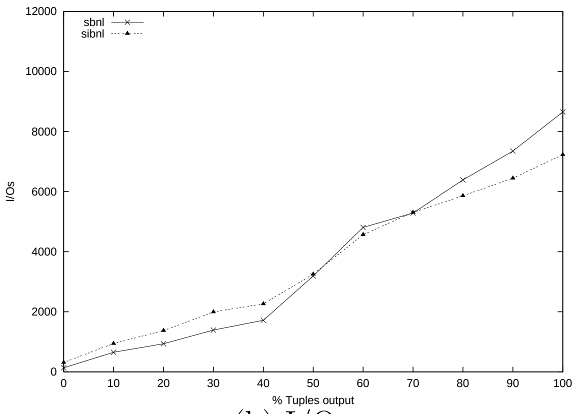
Figure A.4: Comparison of spatial joins (Clustered data) (R100KC5 $\bowtie$ S100KC5)



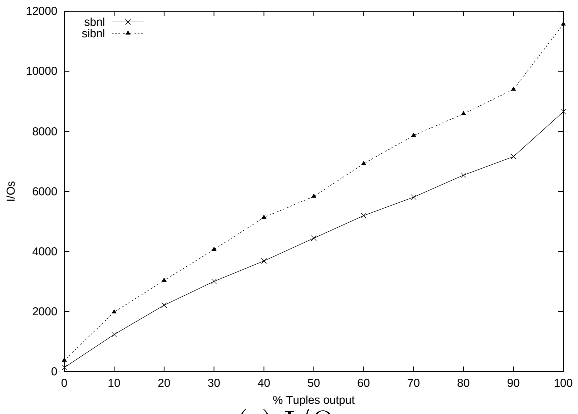
(a) Page Comparison (Clustered)



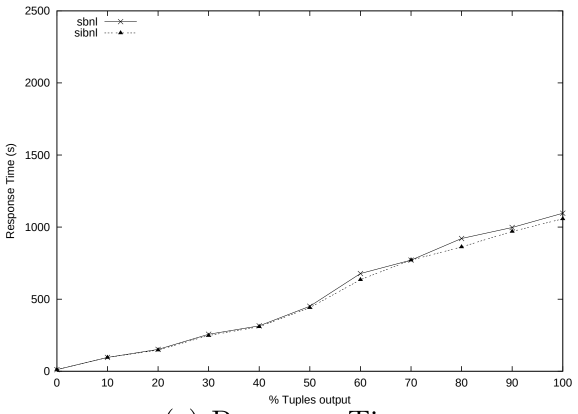
(d) Page Comparison (Shuffled)



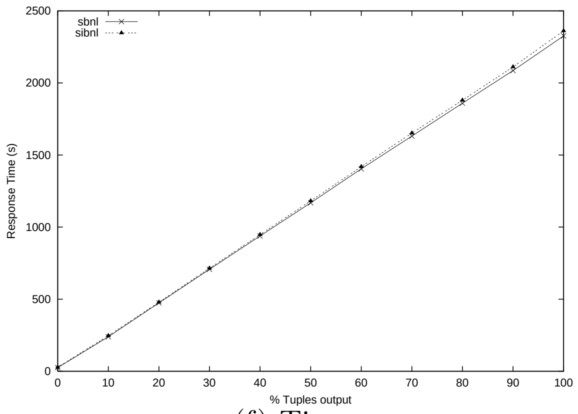
(b) I/Os (Clustered)



(e) I/Os (Shuffled)

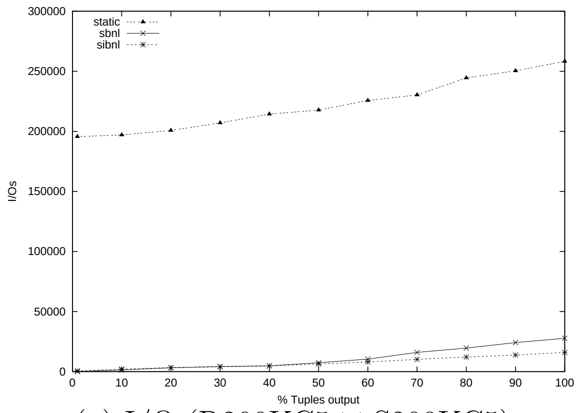


(c) Response Time (Clustered)

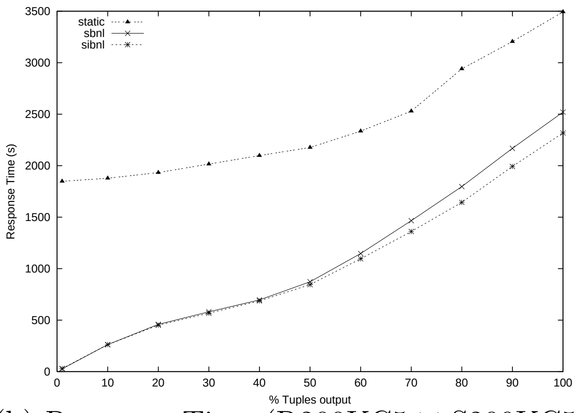


(f) Time (Shuffled)

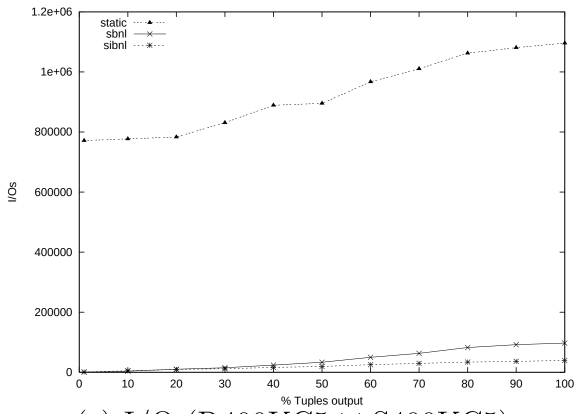
Figure A.5: Clustered vs Shuffled (R100KC5 $\bowtie$ S100KC5)



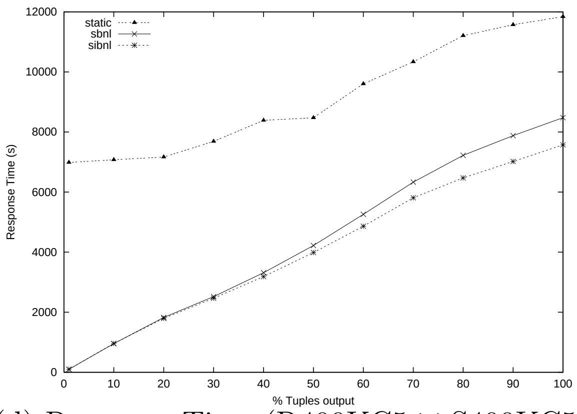
(a) I/O (R200KC5 ⋈ S200KC5)



(b) Response Time (R200KC5 ⋈ S200KC5)

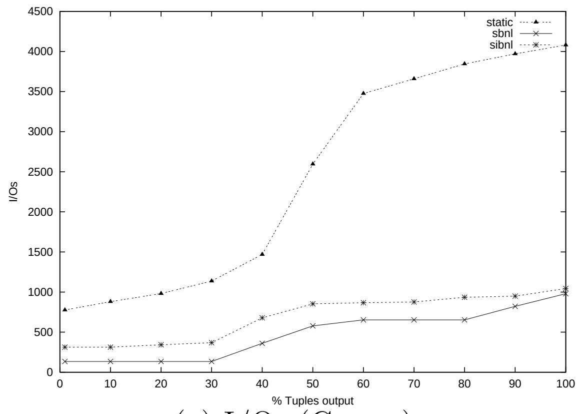


(c) I/O (R400KC5 ⋈ S400KC5)

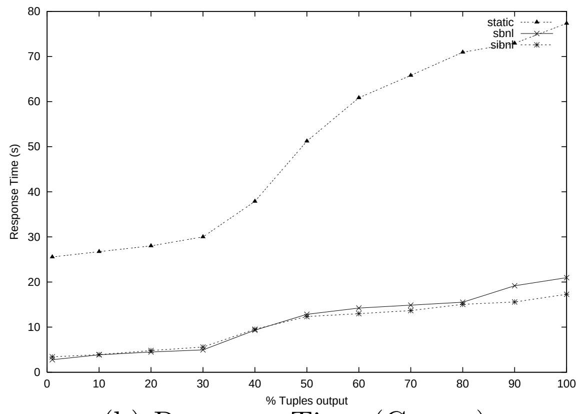


(d) Response Time (R400KC5 ⋈ S400KC5)

Figure A.6: Scalability Test

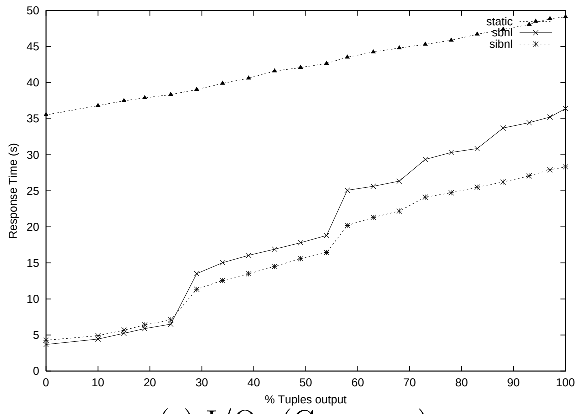


(a) I/Os (Greece)

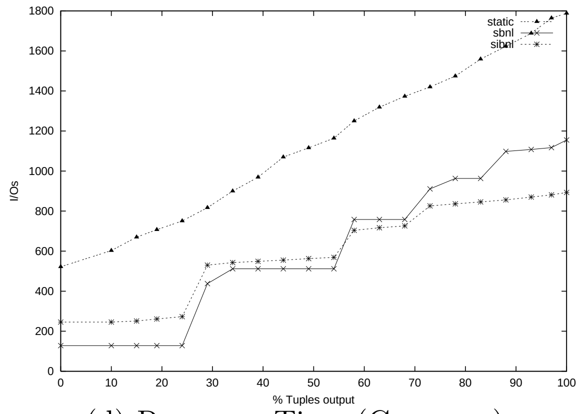


(b) Response Time (Greece)

(Rivers ⋈ Roads)



(c) I/Os (Germany)



(d) Response Time (Germany)

(Railroad Lines ⋈ Roads)

Figure A.7: Performance on Real-Life Data Sets

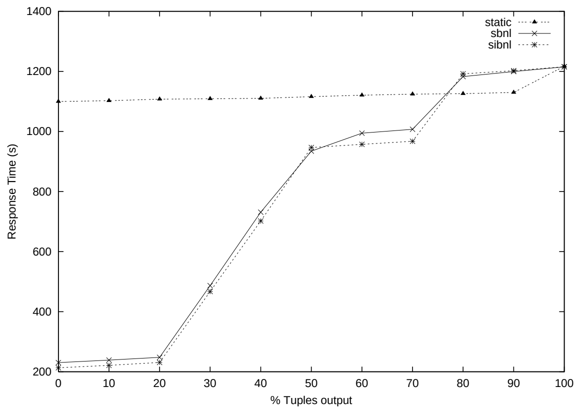


Figure A.8: Poisson Inter-arrival with Means at 2s (R50KC5 $\bowtie$ S50KC5)

Appendix B

XML Data Examples

```
<symbols>
   <symbolTuple>
     <symbol>MSFT</symbol>
     <description>Microsoft</description>
     <market>NasdaqGS</market>
     <industry>Software</industry>
  </symbolTuple>
   <symbolTuple>
     <symbol>GOOG</symbol>
     <description>Google</description>
     <market>NasdaqGS</market>
     <industry>Internet Info Providers</industry>
  </symbolTuple>
  ...
</symbols>
```

(a) Symbol Information (symbol.xml)

```
<quotes>
   <quoteTuple>
     <symbol>MSFT</symbol>
     <shareVolume>75169443</shareVolume>
     <lastSale>26.72</lastSale>
     <currency>USD</currency>
  </quoteTuple>
   <quoteTuple>
     <symbol>GOOG</symbol>
     <shareVolume>6379234</shareVolume>
     <lastSale>443.03</lastSale>
     <currency>USD</currency>
  </quoteTuple>
  ...
</quotes>
```

(b) Stock Quotations (quotes.xml)

Figure B.1: XML Join Scenario A - Stock vs Symbol Information

```
<items>
  <item>
    <title>CNA</title>
    <section>
       <techNews>
         <blurb>Google to bolster privacy of
            online searchers</blurb>
       <keyword>Google</keyword>
       <article>...</article>
     </techNews>
     <techNews>
         <blurb>Japanese researchers unveil
            medical mini robot</blurb>
       <keyword>Robotics</keyword>
       <article>...</article>
     </techNews>
     <businessNews>
     ...
     </businessNews>
    </section>
  </item>
  <item>
    <title>BBC</title>
    ...
  </item>
  ...
</items>
```

(a) News XML (news.xml)

```
<items>
  <blog>
    <name>John Doe</name>
    <entries>
      <entry>
        <entryId>1</entryId>
        <title>Jobs at Google</title>
        <description>...</description>
        <tag>Google</tag>
      </entry>
       ...
    </entries>
  </blog>
</items>
```

(b) Blog XML (blogs.xml)

Figure B.2: XML Join Scenario B - News vs Blog Entries

Appendix C

Danaides System

C.1 Introduction

RSS (Really Simple Syndication) is an XML format used for the publication and syndication of web content. Users subscribe to RSS feeds using RSS readers and aggregators. Although readers and aggregators need to pull and filter data from the RSS feeds at regular intervals, RSS technology implements web data streams.

Existing RSS reader and aggregator software and services provide at most basic keyword-based filtering and simple feed merging. These software and services do not yet support complex queries. Such a support however would enable the utilization of RSS feeds to their full potential of continuous data streams and motivate, in a virtuous circle, the production and consumption of data.

We have designed and implemented a prototype RSS aggregator service, called *Danaïdes*, capable of processing complex queries on continuously updated RSS feeds and of progressively producing results. Users subscribe their queries to the service in a dialect of SQL that can express structured queries, spatial query and similarity queries. The service continuously processes the subscribed queries on the referenced RSS feeds and, in turn, published the query results as RSS feeds. The user can read the result feed in a standard reader software or service or in a dedicated interface.

We demonstrate the prototype and its several user-interfaces with a geographical application using geoRSS feeds. This work is a practical application of our research on progressive query processing algorithms [TB02, TBL06, TBL07c] for data streams.

C.2 Related Work

In [GKL06], the authors describe how commercial databases can be used as a declarative RSS Hub offering structured query capabilities. Since RSS is an XML format it is also natural (yet beyond the scope of the proof of concept that this paper is contributing) to consider XQuery for the formulation of complex query on RSS feeds. In [Iva03], the authors demonstrate the use of XQuery for the filtering and merging of RSS feeds from several blogs.

Whether supporting SQL or XQuery the query processing engines of the new aggregators that we propose must be capable of continuously processing data streams. The above mentioned proposals for complex query in RSS aggregation do not take into account the dynamic and continuous aspect of the RSS feeds. New algorithms are being developed for the processing of queries on data streams. The various algorithms proposed, from the XJoin [UF99] to the Rate-based Progressive Join (RPJ) [TYP$^+$05], Locality-Aware Approximate Sliding Window Join [LCKB06], Progressive Merge Join [DSTW02] and our Result-Rate Based Progressive Join (RRPJ) [TBL07c], try and propose non-blocking solutions that maximize throughput. While [UF99, TYP$^+$05, LCKB06] only consider relational data , our solution [TBL07c] and [DSTW02] can be easily applied to data in other data models.

As far as we know, this is the first proposal for a continuous query processing service for RSS feeds aggregation.

C.3 Scenario and Prototype

The availability of precise, instantaneous, seamless and effortless positioning with the Global Positioning System (GPS), Galileo and GSM triangulation coupled with or embedded in personal and professional portable devices, equipment and gadgets allows the geo-tagging of content created anytime anywhere. From the casual souvenir photographs of a tourist time-stamped, and geo-tagged with longitude, latitude and

Find pairs of earthquake alerts with the same title within 5.6 degree of both latitude and longitude.

```
SELECT *
FROM  rss("http://earthquake.usgs.gov/eqcenter/recenteqsww/catalogs/
eqs1day-M2.5.xml") a, rss("http://earthquake.usgs.gov/eqcenter/
recenteqsww/catalogs/eqs7day-M5.xml") b
WHERE a.title = b.title and
      dist(a.geoLat, a.geoLong, b.geoLat, b.geoLong) < 5.6
```

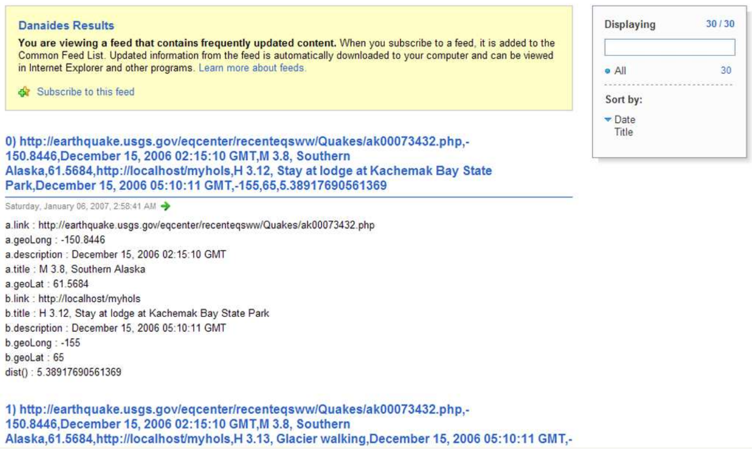
Figure C.1: Sample Query

altitude, published on Flickr [1] to the critical earthquake monitoring data from the U.S. Geological Survey [htta], geo-tagged data is commonly published as RSS feed (A specialization of RSS to publish geographical data is called GeoRSS [httb]).

In this demonstration we show the processing of several complex queries on multiple GeoRSS feeds. We use data from the United States Geological Survey Earthquake Hazards Program [htta]. We show, in particular, queries involving relational joins, spatial joins and similarity join (see Figure C.1). Results are then delivered progressively to the user as a GeoRSS feed. The result feed can be viewed using any RSS reader or aggregator software or service. We use Internet Explorer 7[2] The result feed can also be viewed on a 2D or 3D map. We use a visualization interface that we have developed, which uses Virtual Earth[3] [htt06]. Figure C.2 illustrates these user interfaces.

The *Danaïdes* prototype consists of a scanner and a query processing engine. The scanner periodically pulls data from RSS feeds. The query engine consists of physical algebra operators (e.g. hash join, similarity join, selection, and projection). It constructs a query plan, executes the plan and produces a RSS feed consisting of the results.

[1]Flickr is a trademark of Yahoo! Inc.
[2]Internet Explorer is a trademark of Microsoft Corp.
[3]Virtual Earth is a trademark of Microsoft Corp.



Danaides Results

You are viewing a feed that contains frequently updated content. When you subscribe to a feed, it is added to the Common Feed List. Updated information from the feed is automatically downloaded to your computer and can be viewed in Internet Explorer and other programs. Learn more about feeds.

Subscribe to this feed

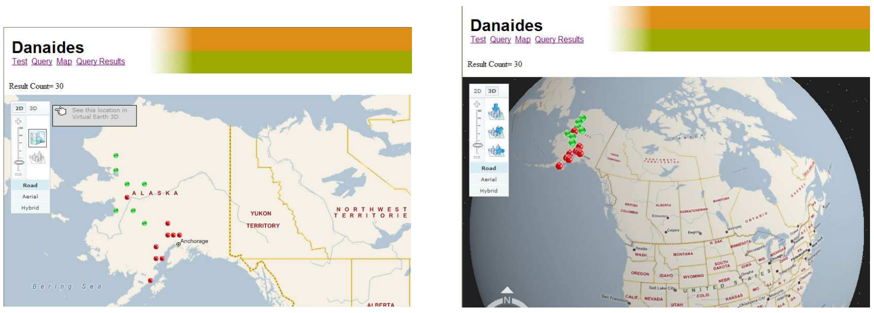
0) http://earthquake.usgs.gov/eqcenter/recenteqsww/Quakes/ak00073432.php,-150.8446,December 15, 2006 02:15:10 GMT,M 3.8, Southern Alaska,61.5684,http://localhost/myhols,H 3.12, Stay at lodge at Kachemak Bay State Park,December 15, 2006 05:10:11 GMT,-155,65,5.38917690561369

Saturday, January 06, 2007, 2:58:41 AM

```
a.link : http://earthquake.usgs.gov/eqcenter/recenteqsww/Quakes/ak00073432.php
a.geoLong : -150.8446
a.description : December 15, 2006 02:15:10 GMT
a.title : M 3.8, Southern Alaska
a.geoLat : 61.5684
b.link : http://localhost/myhols
b.title : H 3.12, Stay at lodge at Kachemak Bay State Park
b.description : December 15, 2006 05:10:11 GMT
b.geoLong : -155
b.geoLat : 65
dist() : 5.38917690561369
```

1) http://earthquake.usgs.gov/eqcenter/recenteqsww/Quakes/ak00073432.php,-150.8446,December 15, 2006 02:15:10 GMT,M 3.8, Southern Alaska,61.5684,http://localhost/myhols,H 3.13, Glacier walking,December 15, 2006 05:10:11 GMT,-

(a) RSS Result Output (Displayed in Internet Explorer 7)



(b) Virtual Earth Augmented with GeoRSS Result

Figure C.2: Various ways of visualizing results from *Danaïdes*

C.4 Summary

In this chapter, we demonstrate the use of our result rate-based progressive algorithm in a system prototype for a RSS aggregator, called *Danaïdes.* Danaides handles the publishing of continuous and progressive complex queries on RSS feeds.

Appendix D

Performance Evaluation of various Sampling Techniques

We study the performance of the several sliding window sampling algorithms when the data distribution changes frequently. We implemented the various window sampling algorithms in C++: (1) Fifo, (2) Reservoir (Res), (3) Expire and (4) Window Reservoir (WinRes) and (5) Chain Sampling (Chain) [BDM02].

The synthetic dataset, D, consists of 500000 tuples. The distribution of the data changes every 50000 tuples (i.e. Every $0.1|D|$). This is achieved by using a zipfian data distribution with Zipfian factor, ζ. For each $0.1|D|$ of data, we randomly generated a ζ factor between 0.0 and 2.0 (inclusive). In addition, to ensure that the skewed values do not cluster within a fixed value range, we also shifted the value ranges for each $0.1|D|$ of data generated.

The results for the experiments are presented in Figure D.1 to Figure D.5. Each figure corresponds to a different window size. The window size is expressed as factor of the dataset size. In each of the figures, we present the results for varying sampling size, which is expressed as a factor of the window size.

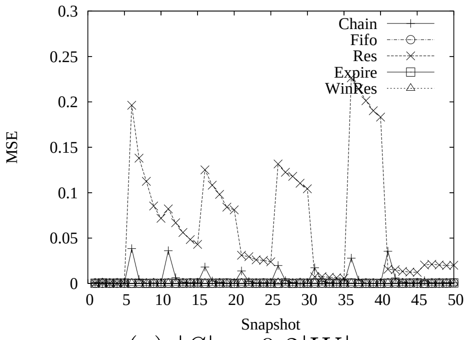
From Figure D.1 to Figure D.5, we can observe that the MSE of the *Res* method is significantly larger. This is because while *Res* is able to maintain a random sample of the entire dataset, it does not ensure that the sample is representative of sliding window of data. Similarly, the *FIFO* method also has high MSE due to its similarity to the *Res* method. The main difference is that instead of randomly replacing a tuple in the reservoir, the *FIFO* method dequeues the first tuple in the FIFO queue and enqueues a newly arrived tuple. In contrast, the other algorithms (*Expire* and *WinRes*) which considers windows of data have relatively small MSE values. The sharp spikes in MSE values corresponds to the points in which the data distribution changes.

In general, if the sample size is equivalent to the window size (Figure D.1(e), D.2(e), D.3(e), D.4(e) and D.5(e)), both *Expire* and *WinRes* have zero MSE.

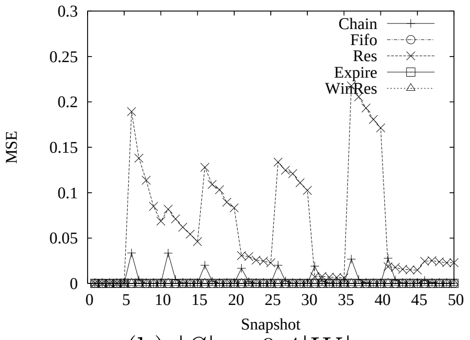
Ordered data

In this experiment, we study the performance of the various window sampling algorithms when the data from Section D are ordered. The synthetic dataset, D, consists of 500000 tuples. The tuples are sorted in ascending order, based on the data values. The results for the experiments are presented in Figure D.6.

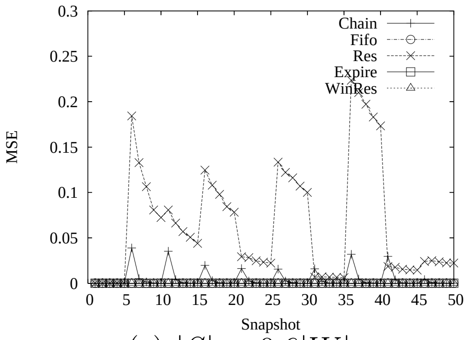
From Figure D.6(a) - (b), we can observe that the performance for all the sampling algorithms shows large MSE values. This is because when the data is ordered and the sample size is small, the sampling algorithms are not able to maintain a uniform sample. However, when the sample size increases, we can observe that except for *Res* (which does not take the sliding window into consideration), the other algorithms are able to perform relatively well (i.e. low MSE values). Similar to the observations from Section D, *Fifo* is sensitive to data distribution changes. This is reflected in the spikes in MSE values in Figure D.6(b), (c), (d) and (e).



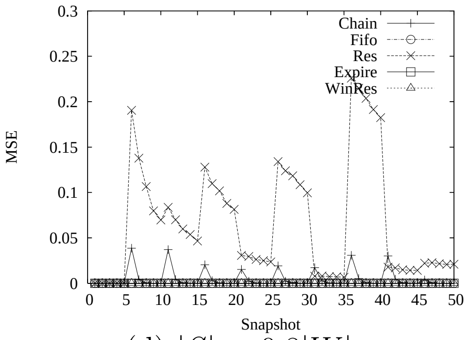
(a) $|S| = 0.2|W|$



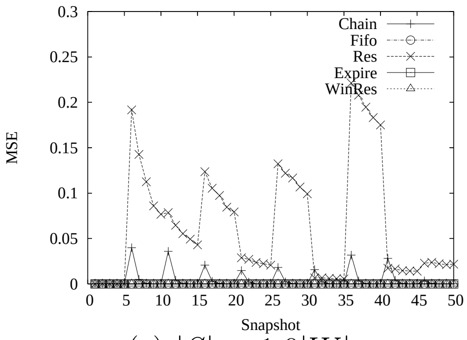
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

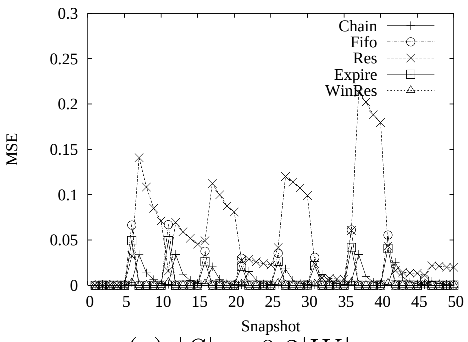


(d) $|S| = 0.8|W|$

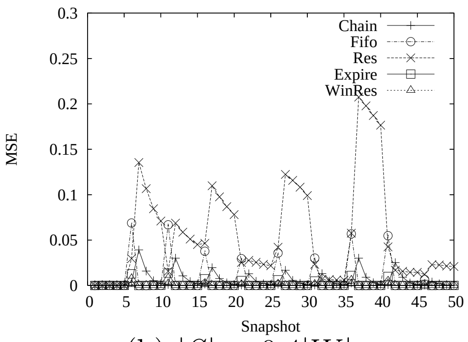


(e) $|S| = 1.0|W|$

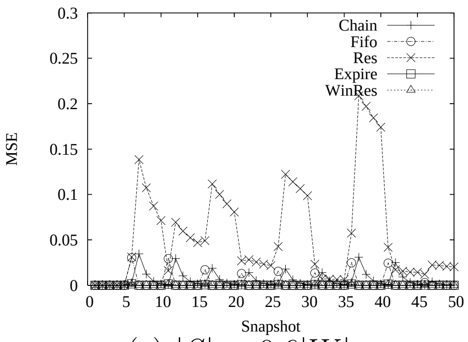
Figure D.1: Varying Zipfian, $|W| = 0.02|D|$ - MSE vs Snapshots



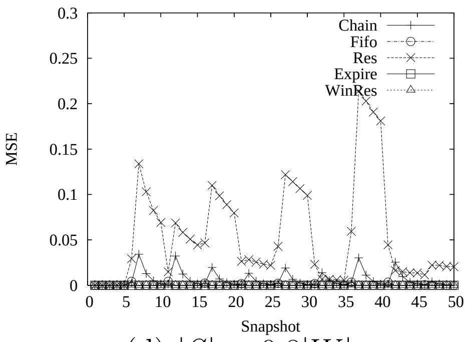
(a) $|S| = 0.2|W|$



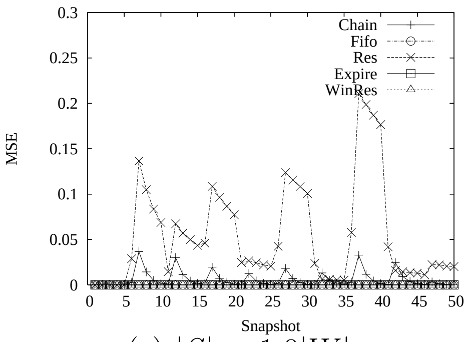
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

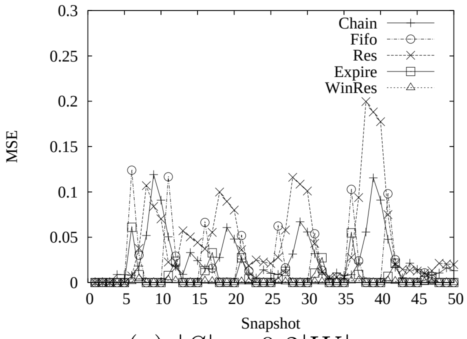


(d) $|S| = 0.8|W|$

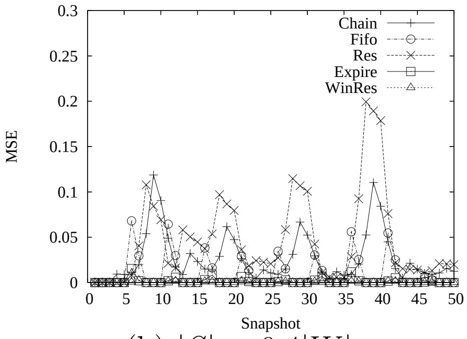


(e) $|S| = 1.0|W|$

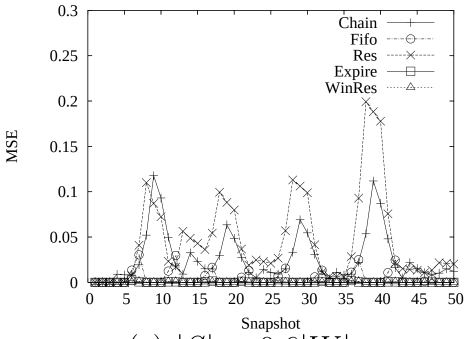
Figure D.2: Varying Zipfian, $|W| = 0.04|D|$ - MSE vs Snapshots



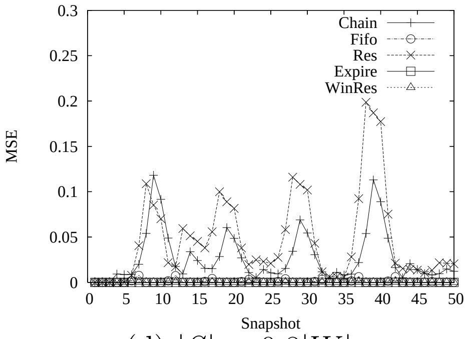
(a) $|S| = 0.2|W|$



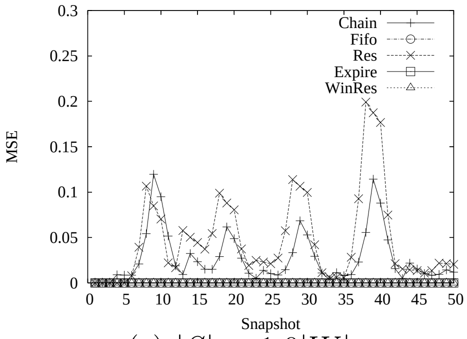
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

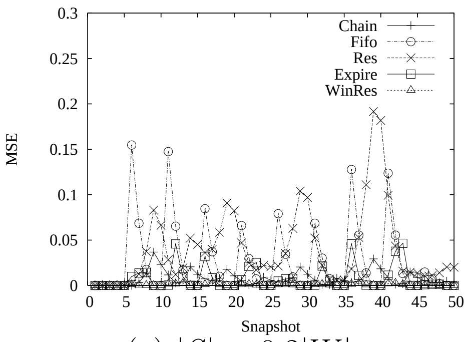


(d) $|S| = 0.8|W|$

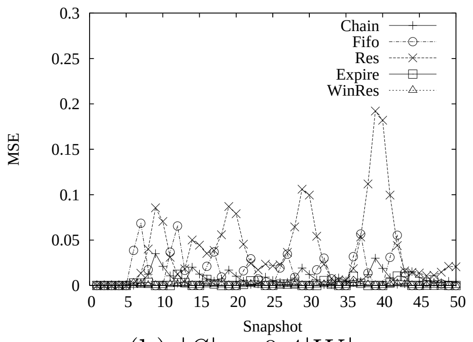


(e) $|S| = 1.0|W|$

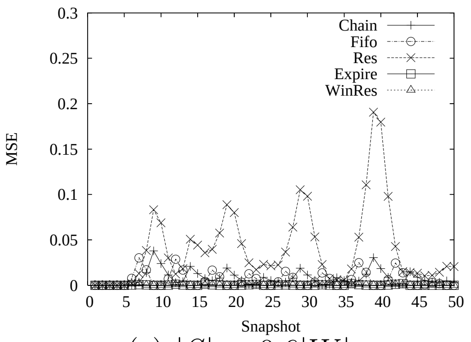
Figure D.3: Varying Zipfian, $|W| = 0.06|D|$ - MSE vs Snapshots



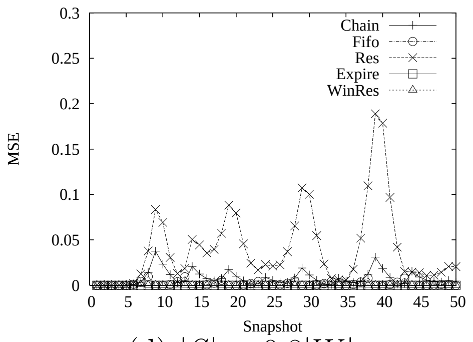
(a) $|S| = 0.2|W|$



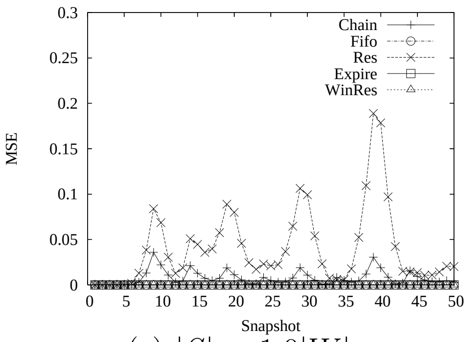
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

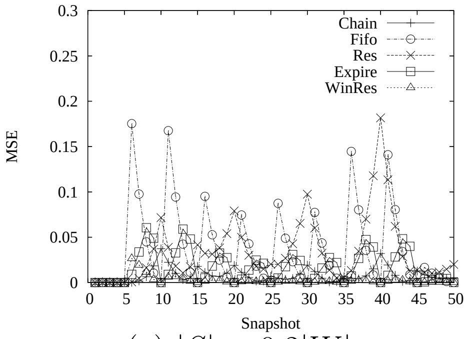


(d) $|S| = 0.8|W|$

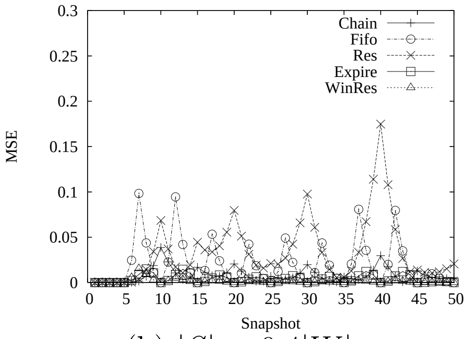


(e) $|S| = 1.0|W|$

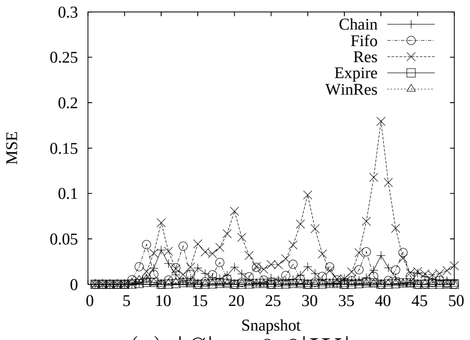
Figure D.4: Varying Zipfian, $|W| = 0.08|D|$ - MSE vs Snapshots



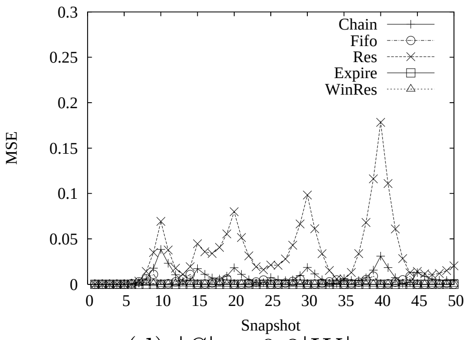
(a) $|S| = 0.2|W|$



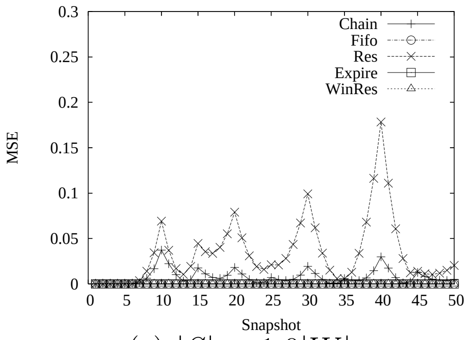
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$

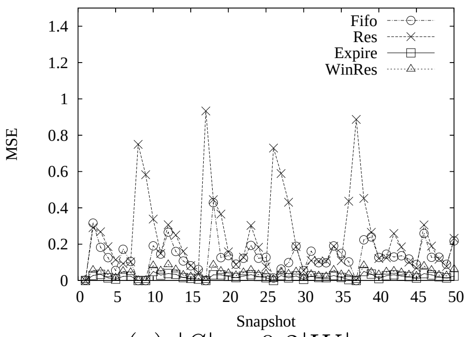


(d) $|S| = 0.8|W|$

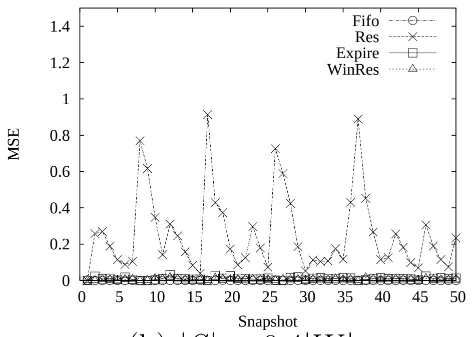


(e) $|S| = 1.0|W|$

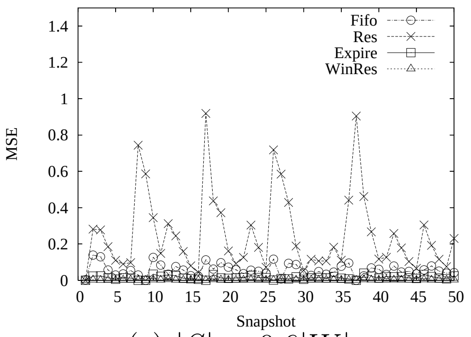
Figure D.5: Varying Zipfian, $|W| = 0.10|D|$ - MSE vs Snapshots



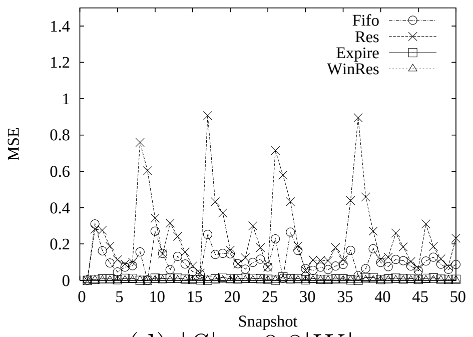
(a) $|S| = 0.2|W|$



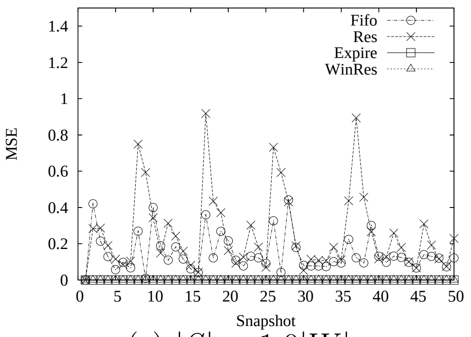
(b) $|S| = 0.4|W|$



(c) $|S| = 0.6|W|$



(d) $|S| = 0.8|W|$



(e) $|S| = 1.0|W|$

Figure D.6: Ordered Dataset, $|W| = 0.02|D|$ - MSE vs Snapshots