Research Report Series

HYBRIDJOIN for Near Real-time Data Warehousing

M. Asif Naeem, Gillian Dobbie, Gerald Weber

TR Number: UoA-SE-2010-2

July 2010

Software Engineering
University of Auckland
Private Bag 92019
Auckland 1
New Zealand
www.se.auckland.ac.nz
info@se.auckland.ac.nz

HYBRIDJOIN for Near Real-time Data Warehousing

M. Asif Naeem
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
mnae006@aucklanduni.ac.nz

Gillian Dobbie
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
gill@cs.auckland.ac.nz

Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
gerald@cs.auckland.ac.nz

Abstract

In the field of real-time data warehousing updates occurring on the source systems need to be reflected in the data warehouse immediately. One important element in real-time data integration is the join of a continuous input data stream with a disk-based relation. For high-throughput streams, stream-based algorithms, such as Mesh Join (MESHJOIN), can be used. However, MESHJOIN cannot deal with intermittent streams, because tuples could wait for an undetermined time, thus defying the real-time character of the stream. The Index Nested Loop Join (INLJ) can be set up so that it processes stream input, and can deal with intermittences in the update stream but it has low throughput. In this paper we introduce a robust stream-based join algorithm called Hybrid Join (HYBRIDJOIN) which combines the two approaches. As a theoretical result we show that HYBRIDJOIN is asymptotically as fast as the fastest of both algorithms. We present performance measurements of our implementation. We use synthetic data, that we base on a Zipfian distribution, which is widely accepted as a plausible distribution for real world identifier sets in many domains. In our experiments, HYBRIDJOIN performs significantly better for typical parameters of the Zipfian distribution, and in general performs in accordance with the theoretical model while the other two algorithms are unacceptably slow under different settings. Hence HYBRIDJOIN is a robust algorithm that generally performs at an acceptable speed.

1 Introduction

Near real-time data warehousing exploits the concepts of data freshness in traditional static data repositories in order to meet the required decision support capabilities. The tools and techniques for promoting these concepts are rapidly evolving [14] [7] [13]. Most data warehouses have already switched from a full refresh [5] [21] [22] to an incremental refresh policy [8] [10] [9]. Further the batch-oriented, incremental refresh approach is moving towards a continuous, incremental refresh approach.

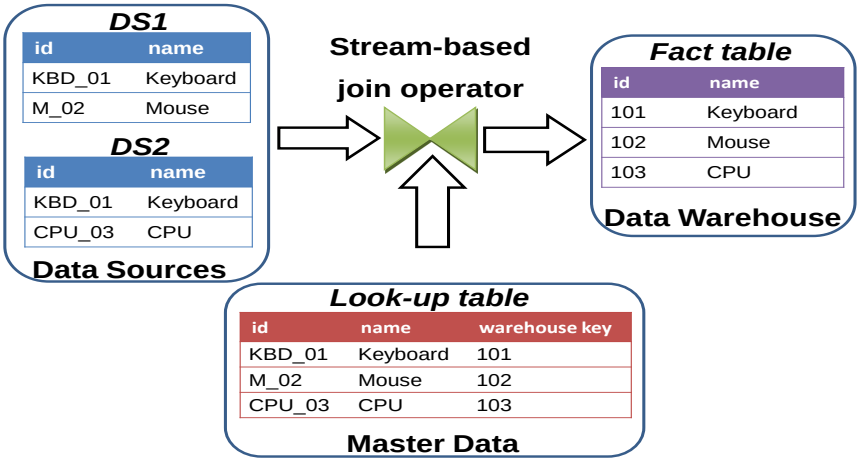


Figure 1: An example of stream-based join

One important research area in the field of data warehousing is data transformation, since the updates coming from the data sources are not in the format required for the data warehouse. Furthermore, in the field of real time data warehousing where a continuous transformation from a source to target format is required, such tasks become more challenging.

In the ETL (Extract-Transform-Load) layer, a number of transformations are performed such as the detection of duplicate tuples, identification of newly inserted tuples, and the enriching of some new attribute values from master data. One common transformation is the key transformation. The key used in the data source may be different from that in the data warehouse and therefore needs to be transformed into the required value for the warehouse key. This transformation can be obtained by implementing a join operation between the update tuples and a lookup table. The lookup table contains the mapping between the source keys and the warehouse keys. Figure 1 shows a graphical interpretation of such a transformation. In the figure, the attributes with column name *id* in both data sources DS_1 and DS_2 contain the source data keys and the attribute with name warehouse key in the lookup table contains the warehouse key value corresponding to these data source keys. Before loading each transaction into the data warehouse each source key is replaced by the warehouse key with the help of a join operator.

In traditional data warehousing the update tuples are buffered in memory and joined when resources become available [20] [17]. Whereas, in real-time data warehousing these update tuples are joined when they are generated in the data sources. One important factor related to the join is that both inputs of the join come from different sources with different arrival rates. The input from the data sources is in the form of an update stream which is fast, while the access rate of the lookup table is comparatively slow due to disk *I/O* cost.

A novel stream-based equijoin algorithm, MESHJOIN [15] [16], was described by Polyzotis et al in *2008*. The MESHJOIN algorithm is in principle a hash join, where the stream serves as the build input and the disk-based relation serves as the probe input. The main contribution is a staggered execution of the hash table build and an optimization of the disk buffer for the disk-based relation.

The algorithm successfully joins the continuous data stream of updates with the slow access rate disk

based relation. However, there are a number of issues that need to be explored further. Firstly, in this approach due to the sequential access of the disk based relation, as the disk *I/O* cost increases the average stay of each stream tuple in the queue also increases. Secondly, the algorithm cannot deal with a bursty update stream efficiently. A detailed explanation of these issues is provided in Section 3.

The Index Nested Loop Join (INLJ) is another algorithm that joins a continuous data stream with the disk based relation, which is capable of dealing with bursty data streams. However, due to being able to process only one stream tuple against the whole disk page, the disk *I/O* cost cannot be amortized over a fast incoming data stream and eventually produces a low service rate.

Based on these observations, we propose a robust stream-based join, called Hybrid Join (HYBRIDJOIN). The key difference between HYBRIDJOIN and MESHJOIN is that HYBRIDJOIN does not read the entire disk relation sequentially but instead accesses it using an index. This not only reduces the stay of every stream tuple in the join window but also minimizes the disk *I/O* cost by guaranteeing that every page read from the disk-based relation is at least used for one stream tuple, while in MESHJOIN there is no guarantee. To amortize the disk read over many stream tuples, the algorithm performs the join of disk pages with all stream tuples currently in memory. This approach guarantees that HYBRIDJOIN is never asymptotically slower than MESHJOIN. In addition, in HYBRIDJOIN, unlike MESHJOIN, the disk load is not synchronised with stream input providing better service rates for bursty streams.

The rest of the paper is structured as follows. The related work is presented in Section 2. Section 3 describes our observations with regard to the current approach. In Section 4 we present the architecture, algorithm, theoretical analysis, cost model, and tuning of our proposed HYBRIDJOIN. The design and implementation of a benchmark for testing HYBRIDJOIN is described in Section 5. The experimental study is discussed in Section 6 and finally Section 7 concludes the paper.

2 Related work

In real-time data warehousing, updates occurring at the source need to be processed in an online fashion. This real-time processing of the update stream introduces the interesting challenges related to the throughput for join algorithms. Some techniques have been introduced already to process join queries over continuous streaming data [4] [2]. In this section we will outline the well known work that has already been done in this area with a particular focus on those which are closely related to our problem domain.

The non-blocking symmetric hash join (SHJ) [20] [19] promotes the proprietary hash join algorithm by generating the join output in a pipeline. In the symmetric hash join there is a separate hash table for each input relation. When the tuple of one input arrives it probes the hash table of the other input, generates a result and stores it in its own hash table. SHJ can produce a result before reading either input relation entirely, however, the algorithm keeps both the hash tables, required for each input, in memory.

The Double Pipelined Hash Join (DPHJ) [6] with a two stage join algorithm is an extension of SHJ. The XJoin algorithm [18] is another extension of SHJ. Hash-Merge Join (HMJ) [12] is also one based on symmetric join algorithm. It is based on push technology and consists of two phases, hashing and merging.

Early Hash Join (EHJ) [11] is a further extension of XJoin. EHJ introduces a new biased flushing policy that flushes the partitions of the largest input first. EHJ also simplifies the strategies to determine the duplicate tuples, based on cardinality and therefore no timestamps are required for arrival and departure of

input tuples. However, because EHJ is based on pull technology, a reading policy is required for inputs.

Mesh Join (MESHJOIN) [15] [16], is designed especially for joining a continuous stream with a disk-based relation for active data warehousing. Although it is an adaptive approach, there are some issues related to the strategy for accessing the disk based relation.

Most recently a partition-based approach [3] was introduced that focuses on minimizing the disk overhead in the MESHJOIN algorithm. However, a switch operator is introduced to switch between the Index Nested Loop Join (INLJ) and MESHJOIN. This switching mode depends on a threshold value for stream tuples in the input buffer. The key component is a wait buffer that holds only join attribute values and maintains them in separate slots with respect to the partitions of the disk based relation. Each disk invocation takes place when either the number of attribute values in any slot of the wait buffer crosses the predefined threshold value or when the whole wait buffer becomes full. We observe that the join attribute values waiting in the slots of the wait buffer, which are not frequent in the input stream, need to wait longer than in the original MESHJOIN algorithm, because the slot does not reach the threshold limit. In addition the author focuses on the analysis of the stream buffer in terms of back log tuples and the delay time rather than analysing the algorithm performance in terms of service rate. Because the author does not provide code for his implementation, we are unable to test this approach practically.

3 Preliminaries and problem definition

In this section we summarize the MESHJOIN and INLJ algorithms with respect to their constraints. At the end of the section we describe the observations that we focus on in this paper.

MESHJOIN was designed to support streaming updates over persistent data in the field of real time data warehousing. The algorithm reads the disk based relation sequentially in segments. Once the last segment is read, it again starts from the first segment. The algorithm contains a buffer, called the disk buffer, to store each segment in memory one at a time, and has a number of memory partitions, equal in size, to store the stream tuples. These memory partitions behave like a queue and are differentiated with respect to the loading time. The number of partitions is equal to the number of segments on the disk while the size of each segment on the disk is equal to the size of the disk buffer. In each iteration the algorithm reads one disk segment into the disk buffer and loads a chunk of stream tuples into the memory partition. After loading the disk segment into memory it joins each tuple from that segment with all stream tuples available in different partitions. Before the next iteration the oldest stream tuples are expired from the join memory and all chunks of the stream are advanced by one step. In the next iteration the algorithm replaces the current disk segment with the next one, loads a chunk of stream tuples into the memory partition, and repeats the above procedure. An overview of MESHJOIN is presented in Figure 2 where we consider only three partitions in the queue, with the same number of pages on disk. For simplicity, we do not consider the hash table at this point and assume that the join is performed directly with the queue.

The crux of the algorithm is that the total number of partitions in the stream queue must be equal to the total number of partitions on the disk and that number can be determined by dividing the size of the disk-based relation R by the size of disk buffer b (i.e. $k=N_R/b$). This constraint ensures that a stream tuple that enters into the queue is matched against the entire disk relation before it expires.

As shown in the figure, for each iteration the algorithm reads a partition of stream tuples, w_i, into the

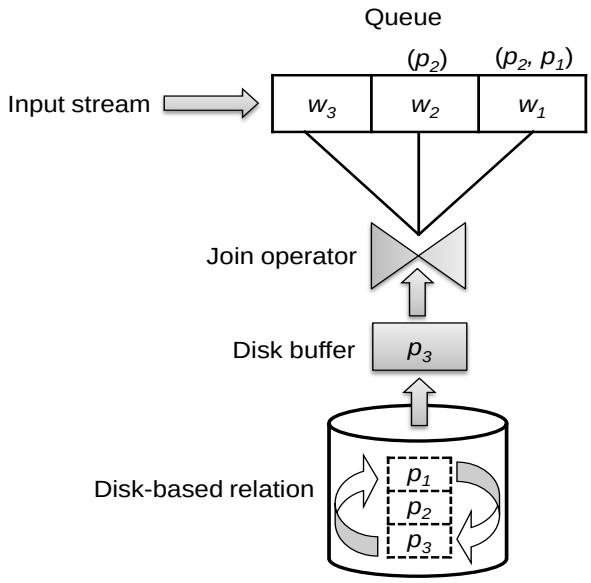


Figure 2: Example of MESHJOIN when disk page *p3* is in memory

queue and one disk page p_j into the disk buffer. At any time *t*, for example when the page p_3 is in memory the status of the stream tuples in the queue can be explained. The w_1 tuples have already joined with the disk pages p_1 and p_2 and therefore after joining with the page p_3 they will be expired. The w_2 tuples have joined only with the page p_2 and therefore, after joining with page p_3 they will advance one step in the queue. Finally, the tuples w_3 have not joined with any disk pages and they will also advance one step in the queue after joining with page p_3. Once the algorithm completes the cycle of *R*, it again starts loading sequentially from the first page.

The MESHJOIN algorithm successfully amortizes the fast arrival rate of the incoming stream by executing the join of disk pages with a large number of stream tuples. However there are still some further issues that exist in the algorithm. Firstly due to the sequential access of *R*, the algorithm reads the unused or less used pages of *R* into memory with equal frequency, which increases the *processing time* for every stream tuple in the queue due to extra disk *I/O*. *Processing time* is the time that every stream tuple spends in the join window from loading to matching without including any delay due to the low arrival rate of the stream. The average *processing time* in the case of MESHJOIN can be estimated using the given formula. Average *processing time (secs)* = $\frac{1}{2}(seektime + accesstime)$ for the whole of *R*

To determine the access rate of disk pages of *R* we performed an experiment using a benchmark that is based on real market economics, the detail is available in Section 5. In this experiment we assumed that *R* is sorted in ascending order with respect to the join attribute value and we measure the rate of use for the same size of segments (each segment contains *20* pages) at different locations of *R*. From the results shown in Figure 3 it is observed that the rate of page use decreases towards the end of *R*. The MESHJOIN algorithm does not consider this factor and reads all disk pages with the same frequency.

Secondly, MESHJOIN cannot deal with bursty input streams effectively. In MESHJOIN a disk invocation occurs when the number of tuples in the stream buffer is equal to or greater than the stream input

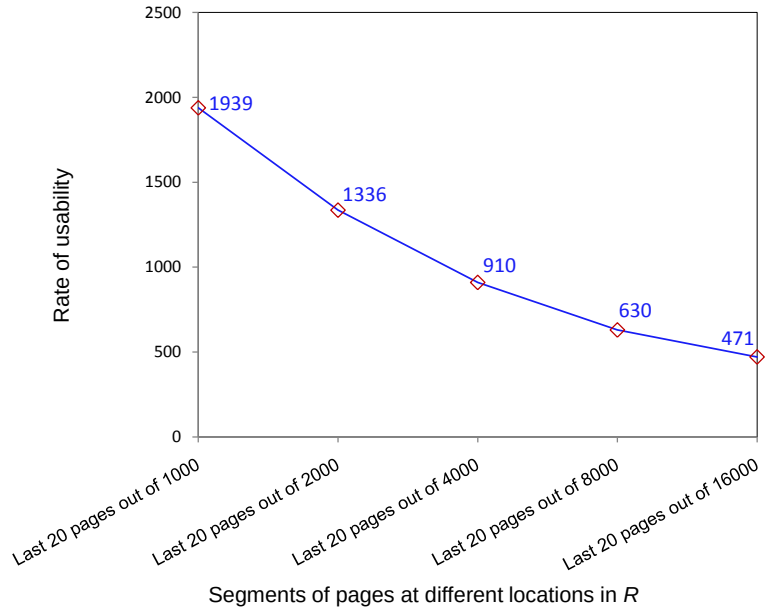


Figure 3: Measured rate of page use at different locations of R while the size of total R is *16000* pages

size w. In the case of intermittent or low arrival rate (λ) of the input stream, the tuples already in the queue need to wait longer due to a disk invocation delay. This *waiting time* negatively affects the performance. The average *waiting time* can be calculated using the given formula.

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

Index Nested Loop Join (INLJ) is another join operator that can be used to join an input stream S with the disk based relation R, using an index on the join attribute. In INLJ for each iteration, the algorithm reads one tuple from S and accesses R randomly with the help of the index. Although in this approach both of the issues presented in MESHJOIN can be handled, the access of R for each tuple of S makes the disk *I/O* cost dominant. This factor affects the ability of the algorithm to cope with the fast arrival stream of updates and eventually decreases the performance significantly.

In summary, the problems that we consider in this paper are: *(a)* the minimization of the *processing time* and *waiting time* for the stream tuples by accessing the disk based relation efficiently, *(b)* dealing with the true nature of a bursty stream.

4 Hybrid Join (HYBRIDJOIN)

In the problem statement we explained our observations related to the MESHJOIN and INLJ algorithms. As a solution to the stated problems we propose a robust stream-based join algorithm called Hybrid Join (HYBRIDJOIN). In this section we describe the architecture, pseudo-code and run time analysis of our proposed algorithm. We also present the cost model that is used for estimating the cost for our algorithm, and for tuning the algorithm.

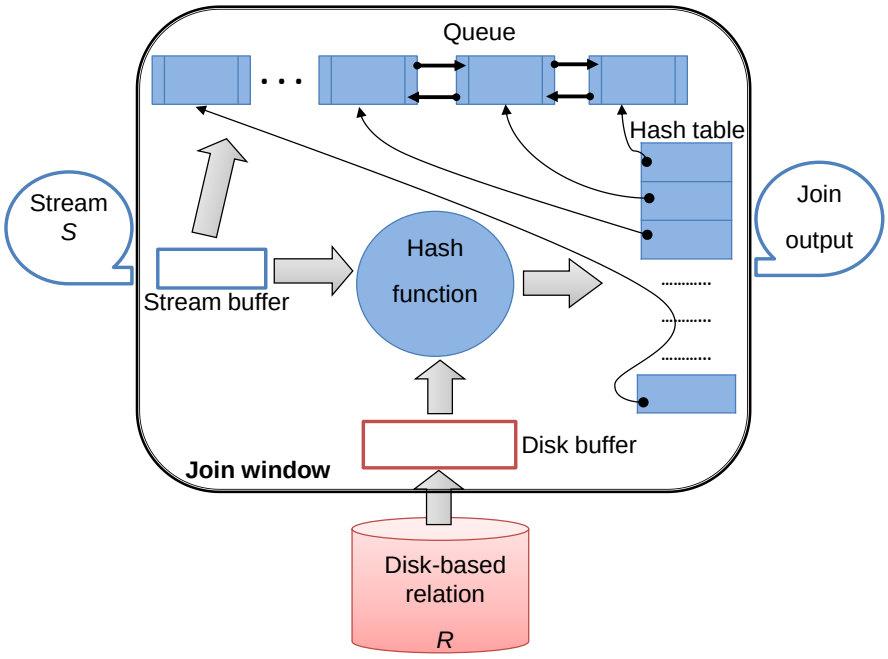


Figure 4: Architecture of HYBRIDJOIN

4.1 Execution architecture

The schematic execution architecture for HYBRIDJOIN is shown in Figure 4. The key components of HYBRIDJOIN are disk buffer, hash table, queue and stream buffer. The disk based relation R and stream S are the inputs. In our algorithm we assume that R is sorted and has an index on the join attribute. The disk page of size v_P from relation R is loaded into the disk buffer in memory. The component queue, based on a double linked list, is used to store the value for join attribute and each node in the queue also contains the addresses of its one step neighbour nodes. Contrary to the queue in MESHJOIN we implement an extra feature of random deletion in our HYBRIDJOIN queue. The hash table is an important component that stores the stream tuples and the addresses of the nodes in the queue corresponding to the tuples. The key benefit of this is when the disk page is loaded into memory using the join attribute value from the queue as an index, instead of only matching one tuple as in INLJ, the algorithm matches the disk page with all the matching tuples in the queue. This helps to amortize the fast arrival stream. In the case where there is a match, the algorithm generates that tuple as an output and deletes it from the hash table along with the corresponding node from the queue while the unmatched tuples in the queue are dealt with in a similar way to the MESHJOIN strategy. The role of the stream buffer is just to hold the fast stream if necessary.

To deal with the intermittencies in the stream, for each iteration the algorithm loads a disk page into memory and checks the status of the stream buffer. In the case where no stream tuples are available in the stream buffer the algorithm will not stop but continues its working until the hash table becomes empty. However, the queue keeps on shrinking continuously and will become empty when all tuples in the hash table are joined. On the other hand when tuples arrive from the stream, the queue again starts growing.

In MESHJOIN every disk input is bound to the stream input while in HYBRIDJOIN we remove this constraint by making each disk invocation independent from the stream input.

HYBRIDJOIN algorithm

Input: A disk based relation R with an index on join attribute and a stream of updates S.
Output: Stream $R \rhd\lhd S$
Parameters: w tuples of S and a page p of R
Method:

```
1. While (true)
2.     Take join attribute value from queue Q.
3.     Read a page p of R into disk buffer using dequeued value as an index.
4.     For each tuple r in pages p
5.         If r ∈ H
6.             Output r ⋈ H
7.             Delete the matched tuple from H and corresponding node from Q.
8.             w ← w+1
9.         EndIf
10.    EndFor
11.    If (stream available)
12.        Read w tuples from stream buffer and load them into H while adding their
           join attribute values into Q.
13.        Reset w to zero.
14.    EndIf
15. EndWhile
```

Figure 5: Pseudo-code for HYBRIDJOIN

4.2 Algorithm

Once the memory is distributed among the join components HYBRIDJOIN starts its execution according to the procedure defined in Figure 5. Normally the join algorithm continues its execution for an infinite amount of time (line 1). In each iteration, the algorithm takes the value of a join attribute from the queue (line 2) and loads a disk page into the disk buffer, using that join attribute value as an index (line 3). After loading the disk page into memory the algorithm reads one by one all tuples from that disk page and probes them in the hash table. In the case of a match, the algorithm generates that tuple as an output and deletes it from the hash table along with the corresponding node from the queue. The algorithm also increments variable w, which contains the next input size for the stream (line 4-10). When the entire disk page is probed the algorithm reads the w tuples from the stream buffer if available, loads them into the hash table, and enqueues their attribute values in the queue. Once the stream input is read the algorithm resets the value of w to zero (line 11-13).

4.3 Asymptotic runtime analysis

We compare the asymptotic runtime of HYBRIDJOIN with that of MESHJOIN and INLJ as throughput, i.e. the time needed to process a stream section. The throughput is the inverse of the service rate. Consider the time for a concrete stream prefix s. We denote the time needed to process stream prefix s as *MEJ(s)* for MESHJOIN, as *INLJ(s)* for index join, and as *HYJ(s)* for HYBRIDJOIN. Every stream prefix represents a binary sequence, and by viewing this binary sequence as a natural number, we can apply asymptotic complexity classes to the functions above. Note therefore that the following theorems do not use functions on input lengths, but on concrete inputs. The resulting theorems imply analogous asymptotic behavior on

input length, but are stronger than statements on input length. We assume that the setup for HYBRIDJOIN and for MESHJOIN is such that they have the same number H_t of stream tuples in the hash table - and in the queue accordingly.

Comparison with MESHJOIN:

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

Proof: To prove the theorem, we have to prove that HYBRIDJOIN performs no worse than MESHJOIN. The cost of MESHJOIN is dominated by the number of accesses to *R*. For asymptotic runtime, random access of disk pages is as fast as sequential access (seek time is only a constant factor). For MESHJOIN with its cyclic access pattern for *R*, every page of *R* is accessed exactly once after every H_t stream tuples. We have to show that for HYBRIDJOIN no page is accessed more frequently. For that we look at an arbitrary page *p* of *R* at the time it is accessed by HYBRIDJOIN. The stream tuple at the front of the queue has some position *i* in the stream. There are H_t stream tuples currently in the hash table, and the first tuple of the stream that is not yet read into the hash table has position $i+H_t$ in the stream. All stream tuples in the hash table are joined against the disk-based master data tuples on *p*, and all matching tuples are removed from the queue. We now have to determine the earliest time that *p* could be loaded again by HYBRIDJOIN. For *p* to be loaded again, a stream tuple must be at the front of the queue, and has to match a master data tuple on *p*. The first stream tuple that can do so is the aforementioned stream tuple with position $i+H_t$, because all earlier stream tuples that match data on *p* have been deleted from the queue. This proves the theorem.

Comparison with INLJ:

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

Proof: INLJ performs a constant number of disk accesses per stream tuple. For the theorem it suffices to prove that HYBRIDJOIN performs no more than a constant number of disk accesses per stream tuple as well. We consider first those stream tuples that remain in the queue until they reach the front of the queue. For each of these tuples, HYBRIDJOIN loads a part of *R* and hence makes a constant number of disk accesses. For all other stream tuples, no separate disk access is made. This proves the theorem.

4.4 Cost model

In this section we derive the general formulas to calculate the cost for our proposed HYBRIDJOIN. We generally calculate the cost in terms of memory and processing time. Equation (1) describes the total memory used to implement the algorithm (except the stream buffer). Equation (3) calculates the processing cost for *w* tuples while the average size for *w* can be calculated using Equation (2). Once the processing cost for *w* tuples is measured, the service rate μ can be calculated using Equation (4). The symbols used to measure the cost are specified in Table 1.

4.4.1 Memory cost

In HYBRIDJOIN, the maximum portion of the total memory is used for the hash table *H* while a comparatively smaller amount is used for the disk buffer and the queue. We can easily calculate the size for each of them separately.

Memory reserved for the disk buffer *(bytes)* $= v_P$

Table 1: Notations used in cost estimation of HYBRIDJOIN

Parameter name	**Symbol**
Total allocated memory *(bytes)*	M
Memory reserved by HYBRIDJOIN *(bytes)*	M_{HYBRID}
Stream arrival rate *(tuples/sec)*	λ
Service rate *(processed tuples/sec)*	μ
Average stream input size *(tuples)*	w
Stream tuple size *((bytes))*	v_S
Size of disk buffer *(bytes)* = Size of disk page	v_P
Size of disk tuple *(bytes)*	v_R
Size of disk buffer *(tuples)*	$D_t = \frac{v_P}{v_R}$
Memory weight for hash table	α
Memory weight for queue	β
Size of hash table *(bytes)*	H_m
Size of hash table *(tuples)*	$H_t = \frac{H_m}{v_S}$
Size of queue *(bytes)*	Q_m
Size of disk based relation *R (tuples)*	R_t
Exponent value for benchmark	e
Cost to read one disk page into disk buffer *(nanosecs)*	$c_{I/O}(v_P)$
Cost of removing one tuple from *H* and *Q (nanosecs)*	c_E
Cost of reading one stream tuple into the stream buffer *(nanosecs)*	c_S
Cost of appending one tuple into *H* and *Q (nanosecs)*	c_A
Cost of probing one tuple into the hash table *(nanosecs)*	c_H
Cost to generate the output for one tuple *(nanosecs)*	c_O
Total cost for one loop iteration of HYBRIDJOIN *(secs)*	c_{loop}

Memory reserved for the hash *(bytes)*, $H_m = \frac{\alpha}{\alpha+\beta}(M - v_P)$
Memory reserved for the queue *(bytes)*, $Q_m = \frac{\beta}{\alpha+\beta}(M - v_P)$
The total memory used by HYBRIDJOIN can be determined by aggregating all the above.

$$M_{HYBRID} = v_P + \frac{\alpha}{\alpha+\beta}(M - v_P) + \frac{\beta}{\alpha+\beta}(M - v_P) \tag{1}$$

Currently we are not including the memory reserved for the stream buffer due to its small size (*0.05 MB* was sufficient in all our experiments).

4.4.2 Processing cost

In this section we calculate the processing cost for HYBRIDJOIN. To calculate the processing cost it is necessary to calculate the average stream input size, *w*, first.

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

- Size of hash table H_t
- Size of disk buffer D_t
- Size of disk based relation R_t
- The exponent value for benchmark *e*

In our experiments *w* is directly proportional to the size of the hash table H_t and the size of the disk buffer D_t, and is inversely proportional to the size of disk based relation R_t and the exponent value *e* for benchmark. The fourth parameter represents the market economics, explained in Section 5, and by using the value *-1* we can approximately model the *80/20* Rule [1] for market sales. Therefore, the formula for *w* is:

$$w \propto \frac{H_t \cdot D_t}{(-1)R_t}$$

$$w = -k\frac{H_t \cdot D_t}{R_t} \tag{2}$$

where *k* is a constant whose value in our settings is *-1.36*.

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

$$c_{loop} = 10^{-9}[c_{I/O}(v_P) + \frac{v_P}{v_R}c_H + w(c_O + c_E + c_S + c_A)] \tag{3}$$

Since in each c_{loop} seconds, the algorithm processes *w* tuples of stream *S*, the service rate μ can be calculated by dividing *w* by the cost for one loop iteration.

$$\mu = \frac{w}{c_{loop}} \tag{4}$$

4.5 Tuning

Tuning of the join components is important to make efficient use of available resources. In HYBRIDJOIN the disk buffer is the key component to tune to amortize the disk *I/O* cost on fast input data streams. From Equation (4) the service rate depends on *w* and the cost c_{loop}, required to process these *w* tuples. In HYBRIDJOIN for a particular setting $(M = 50MB)$ assuming the size of *R* and the exponent value are fixed ($R_t = 2Million$ and $e = -1$), from Equation (2) *w* then depends on the size of hash table and the size of disk buffer. Furthermore the size of hash table also dependent on the size of the disk buffer as shown in Equation (1). Therefore, using Equations (2), (3) and (4) the service rate μ can be specified as a function of v_P and the value for v_P at which the service rate is maximum can be determined by applying standard calculus rules.

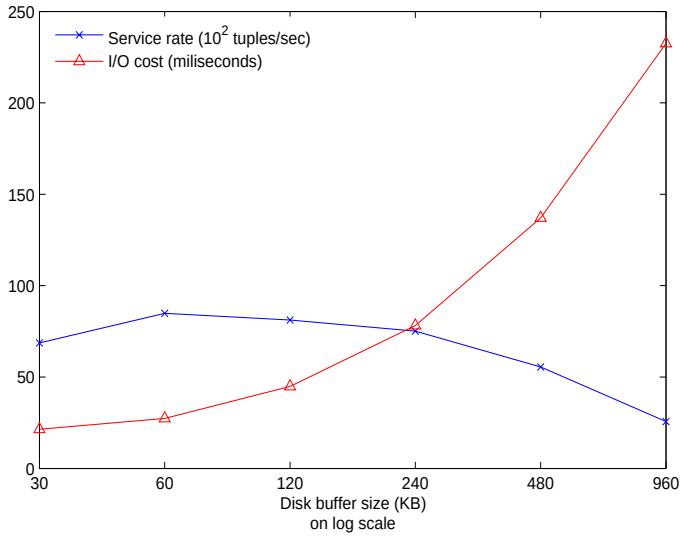


Figure 6: Tuning of disk buffer

In order to explain it experimentally, Figure 6 shows the relationship between the *I/O* cost and service rate. From the figure it can be observed that in the beginning, for a small disk buffer size, the service rate is also small because there are fewer matching tuples in the queue. In other words we can say w is also small. However, the service rate increases with an increase in the size of the disk buffer due to more matching tuples in the queue. After reaching a particular value of the disk buffer size the trend changes and performance decreases with further increments in the size of the disk buffer. The plausible reason behind this decrease is the rapid increment in the disk *I/O* cost.

5 Tests with locality of disk access

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

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

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

The popular types of distributions are Zipfian distributions, which exhibit a power law similar to Zipf's law. Zipfian distributions are discussed as at least plausible models for sales [1], where some products are sold frequently while most are sold rarely. If we generate a distribution using a power law with exponent *-1* it approximately models the *80/20* Rule [1] i.e. *80%* of the sales are from *20%* of the products.

We therefore designed a generator for synthetic data that follows a Zipfian distribution, and use this to demonstrate that HYBRIDJOIN performance increases through locality, and that HYBRIDJOIN outperforms MESHJOIN.

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

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

This bursty generation of tuples models a flow of sales transactions which depends upon fluctuations over several time periods, such as market hours, weekly rhythms and seasons.

The pseudo-code for the generation of our proposed benchmark is shown in Figure 7. In the figure $S_{TREAM}G_{ENERATOR}$ is the main procedure while $G_{ET}D_{ISTRIBUTION}V_{ALUE}$ and $S_{WAP}S_{TATUS}$ are the sub-procedures that are called from the main procedure. According to the main procedure a number of virtual stream objects (in our case *10*), each representing the same distribution value obtained from the $G_{ET}D_{ISTRIBUTION}V_{ALUE}$ procedure, are inserted into a priority queue, which always keeps sorting these objects into ascending order (line 5 to 7). Once all the virtual stream objects are inserted into the priority queue the top most stream object is taken out (line 8). To generate an infinite stream a loop is executed (line 9 to 18). In each iteration of the loop, the algorithm waits for a while (depending on the value of variable *oneStep*) and then checks whether the current time is greater than the time when that particular object was inserted. If the condition is true the algorithm dequeues the next object from the priority queue and calls the $S_{WAP}S_{TATUS}$ procedure (line 11 to 14). The $S_{WAP}S_{TATUS}$ procedure enqueues the current dequeued stream object by updating its time interval and bandwidth (line 19 to 27). Once the value of the variable *totalCurrentBandwidth* is updated, the main procedure generates the final stream tuple values as an output using the procedure $G_{ET}D_{ISTRIBUTION}V_{ALUE}$ line (15 to 17). For each call to procedure $G_{ET}D_{ISTRIBUTION}V_{ALUE}$, it returns the random value by implementing Zipf's law with exponent value equal to *-1*(line 28 to 31).

The experimental representation of our benchmark is shown in Figure 8 and Figure 9, while the environment in which the experiments are conducted is described in Section 6.1. As described earlier in this section, our benchmark is based on two characteristics, one is the frequency of selling each product while the other is the flow of these sales transactions. Figure 8 validates the first characteristic about real market sales. In the figure the *x-axis* represents the variety of products while the *y-axis* represents the sales. Therefore, from the figure it can be observed that only a limited number of products (*20%*) are sold frequently while the rest of the products are rarely sold.

Our proposed HYBRIDJOIN is fully adapted to such kinds of realistic benchmarks in which only a small portion of *R* is accessed again and again while the rest of *R* is accessed rarely.

Figure 9 represents the flow of transactions, which is the second characteristic of our benchmark. From the figure it is clear that the flow of transactions varies with time and is bursty rather than appearing at a regular rate.

Procedure *StreamGenerator*

```
totalCurrentBandwidth ←0
timeInChosenUnit ←o
on ←false
d←GetDistributionValue( )
For i=1 to N
  PriorityQueue.enqueue(d,bandwidth= Math.power(2,i),timeInChosenUnit=currentTime( ))
EndFor
current←PriorityQueue.dequeue( )
While (true)
Wait(oneStep)
 If (currentTime( ) > current.timeInChosenUnit)
   current←PriorityQueue.dequeue( )
   SwapStatus(current)
 EndIf
 For j=1 to totalCurrentBandwidth
   Output GetDistributionValue( )
 EndFor
EndWhile
```

Procedure *SwapStatus (current)*

```
timeInChosenUnit←(current.timeInChosenUnit +getNextRandom() × oneStep ×  current.bandwidth)
If (on)
  totalCurrentBandwidth ←totalCurrentBandwidth – current.bandwidth
  on←false
Else
  totalCurrentBandwidth ←totalCurrentBandwidth +current.bandwidth
  on←true
EndIf
PriorityQueue.enqueue(current)
```

Procedure *GetDistributionValue*

$$sumOfFrequency \leftarrow \int \frac{1}{x} dx_{at\,x=\max} - \int \frac{1}{x} dx_{at\,x=\min}$$

$$random \leftarrow getNextRandom()$$

$$distributionValue \leftarrow inverseIntegralOf\left(random \times sumOfFrequency + \int \frac{1}{x} dx_{at\,x=\min}\right)$$

return $\lfloor distributionValue \rfloor$

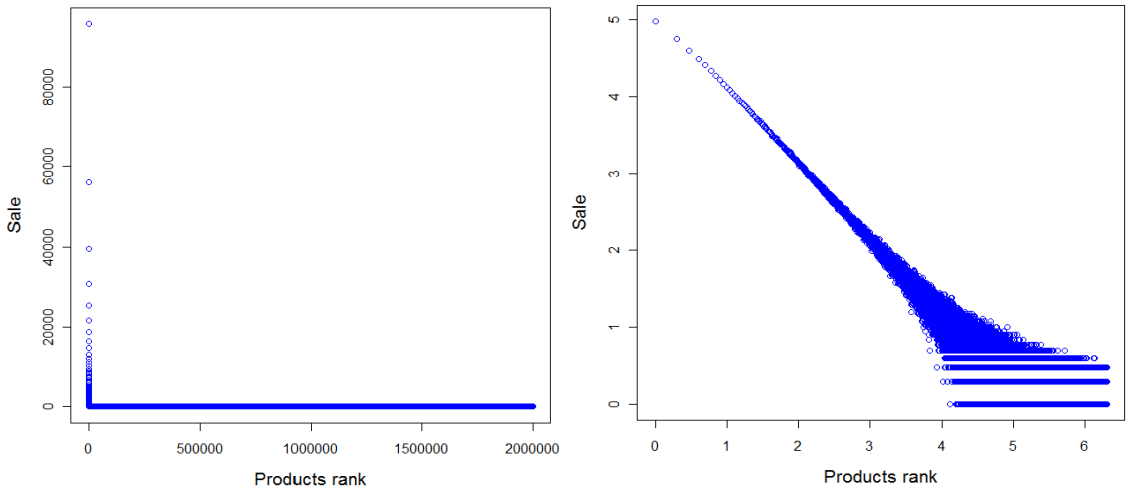
Figure 7: Pseudo-code for benchmark

6 Experiments

We performed an extensive experimental evaluation of the HYBRIDJOIN, proposed in section 4, on the basis of synthetic datasets. In this section we illustrate the environment of our experiments and analyze the results that we obtained using different scenarios.

6.1 Experimental arrangement

In order to implement the prototypes of existing MESHJOIN, Index Nested Loop Join(INLJ) and our proposed HYBRIDJOIN algorithms we used the following hardware and data specifications.



(a) On plain scale

(b) On log scale (both axis are on log scale)

Figure 8: A long tail distribution using Zipf's law that implements *80/20* Rule

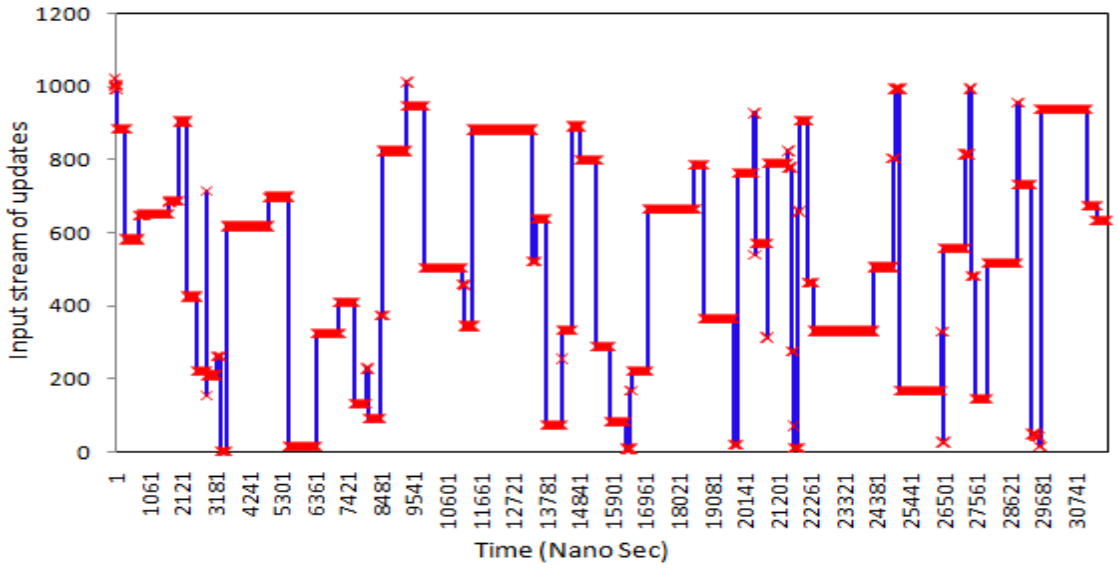


Figure 9: An input stream having bursty and self-similarity type of characteristics

Hardware specifications: We carried out our experimentation on a *Pentium-IV 2×2.13GHz* machine with *3G* main and *160G* disk memory under *Windows-XP*. We implemented the experiment in *Java* using the *Eclipse IDE Version: 3.3.1.1*. We also used built-in plugins, provided by *Apache*, and *nanoTime()*, provided by the *Java API*, to measure the memory and processing time respectively.

Data specifications: We analyzed the performance of each of the algorithms using synthetic data. The relation *R*, master data, is stored on disk using *MySQL version 5.0* database, while the bursty type of stream data is generated at run time using our own benchmark algorithm. Both the algorithms read master data from the database.

In transformation, join is normally performed between the *primary key* (key in lookup table) and the *foreign key* (key in stream tuple) and therefore our HYBRIDJOIN supports join for both one-to-one and

Table 2: Data specification

Parameter	**value**
Disk-based data	
Size of disk-based relation R	0.5 *millions* to 08 *millions tuples*
Size of each tuple	120 *bytes*
Stream data	
Size of each tuple	20 *bytes*
Size of each node in queue	12 *bytes*
Stream arrival rate λ	125 to 2000 *tuples/sec*
Benchmark	
Based on	Zipf's law
Characteristics	Bursty and self-similar

one-to-many relationships. In order to implement the join for one-to-many relationships it needs to store multiple values in the hash table against one key value. However the hash table provided by the *Java API* does not support this feature therefore, we used *Multi-Hash-Map*, provided by *Apache*, as the hash table in our experiments. The detailed specification of the data set that we used for analysis is shown in Table 2.

Measurement strategy: The performance or service rate of the join is measured by calculating the number of tuples processed in a unit second. In each experiment the algorithm runs for one hour and we start our measurements after *20* minutes and continue it for *20* minutes. For more accuracy we take three readings for each specification and then calculate confidence intervals for every result by considering 95% accuracy. Moreover, during the execution of the algorithm no other application is assumed to run in parallel.

6.2 Experimental results

We conducted our experiments in two dimensions. In Section 6.2.1 we compare the performance of all three approaches, while in Section 6.2.2 we validate the cost by comparing it with the predicted cost.

6.2.1 Performance comparison

As the source for MESHJOIN is not openly available, we implemented the MESHJOIN algorithm ourselves. In our experiments we compare the performance in two different ways. First, we compare HYBRIDJOIN with MESHJOIN with respect to the time, both *processing time* and *waiting time*. Second, we compare the performance in terms of service rate with other two algorithms.

Performance comparisons with respect to time: To test the performance with respect to time we conduct two different experiments. The experiment, shown in Figure 10(a), presents the comparisons with respect to the *processing time* on a log scale, while Figure 10(b) depicts the comparisons with respect to *waiting time*. The terms *processing time* and *waiting time* have already been defined in Section 3. According to Figure 10(a) the *processing time* in the case of HYBRIDJOIN is significantly smaller than that of MESHJOIN. The reason behind this is the different strategy to access R. The MESHJOIN algorithm accesses all disk pages with the same frequency without considering the rate of use of each page on the

disk. In HYBRIDJOIN an index-based approach is implemented to access R that never reads unused disk pages. In this experiment we do not reflect the *processing time* for INLJ because it is constant even when the size of R changes.

In the experiment shown in Figure 10(b) we compare the time that each algorithm waits (except Index Nested Loop Join). In the case of INLJ, since the algorithm works at tuple level, the algorithm does not need to wait but this delay then appears in the form of stream backlog that occurs due to faster incoming stream rate than the processing rate. The ratio of this delay (*waiting time*) increases exponentially with an increase in the stream arrival rate.

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

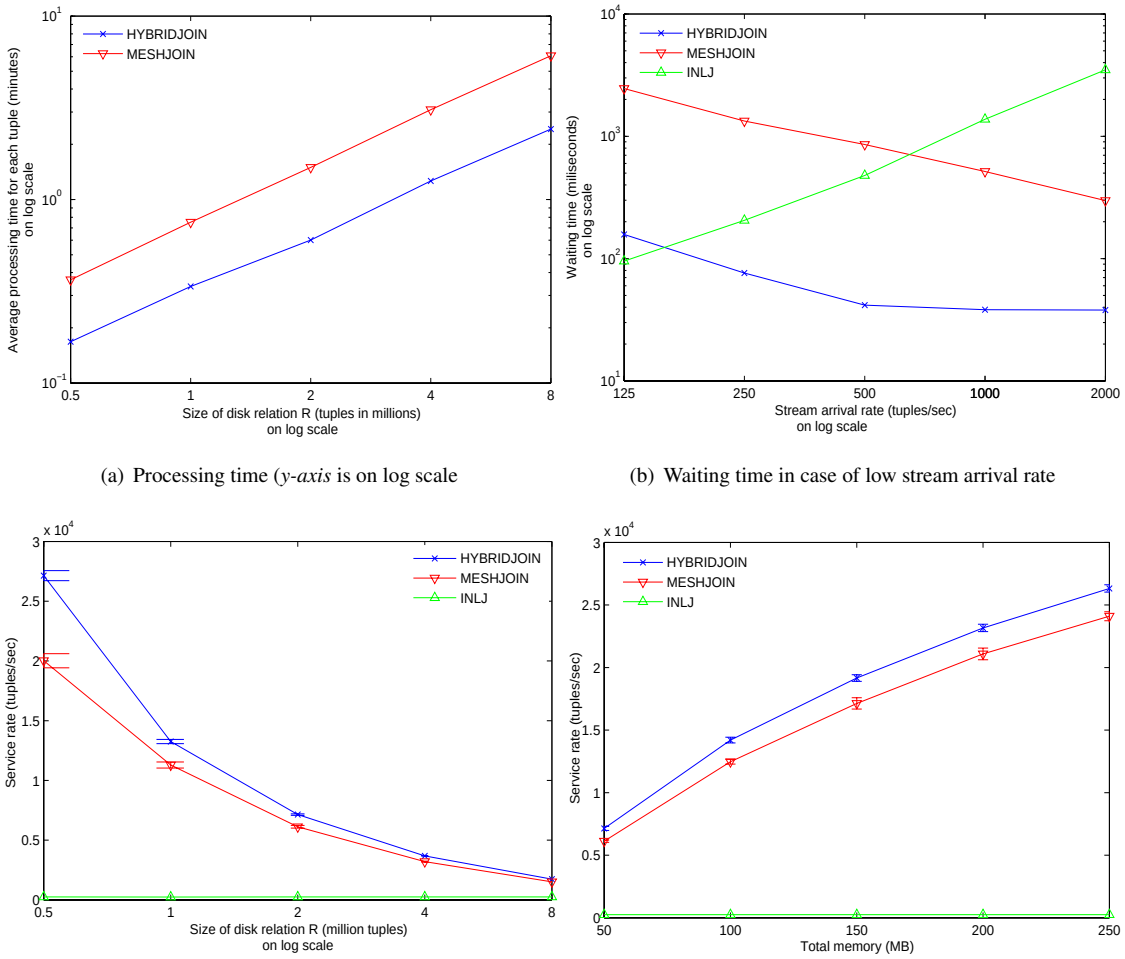
Performance comparisons with respect to service rate: In this category of our experiments we compare the performance of HYBRIDJOIN in terms of the service rate with the other two join algorithms by varying both the total memory budget and the size of R with a bursty stream. In the experiment shown in Figure 10(c) we assume the total allocated memory for the join is fixed while the size of R varies exponentially. From the figure it can be observed that for the small size of R, the performance of HYBRIDJOIN is significantly better compared with the other join approaches. However, this factor of improvement decreases with an increase in the size of relation R. The reason is that by increasing the size of R the probability of matching the stream tuples against the disk page decreases while the disk *I/O* cost remains the same because of the fixed size of the disk page. In our second experiment of this category we analyse the performance of HYBRIDJOIN using different memory budgets, while the size of R is fixed (*2* million tuples). Figure 10(d) depicts the comparisons of the approaches. From the figure it is clear that for all memory budgets the performance of HYBRIDJOIN is better as compared to the other two algorithms.

6.2.2 Cost validation

In this experiment we validate the cost model for all three approaches by comparing the predicted cost with the measured cost. Figure 11 presents the comparisons of both costs. In the figure it is demonstrated that the predicted cost closely resembles the measured cost in every approach which is an evidence for our accurate implementations.

7 Conclusions and future work

In the context of real-time data warehousing a join operator is required to perform a continuous join between the fast stream and the disk based relation within limited resources. In this paper we investigated two available stream-based join algorithms and presented a robust join algorithm, HYBRIDJOIN. Our main objectives in HYBRIDJOIN are: *(a)* to minimize the stay of every stream tuple in the join window by improving the efficiency of the access to the disk based relation, *(b)* to deal with the true nature of update streams. We developed a cost model and tuning methodology in order to achieve the maximum perfor-



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

(b) Waiting time in case of low stream arrival rate

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

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

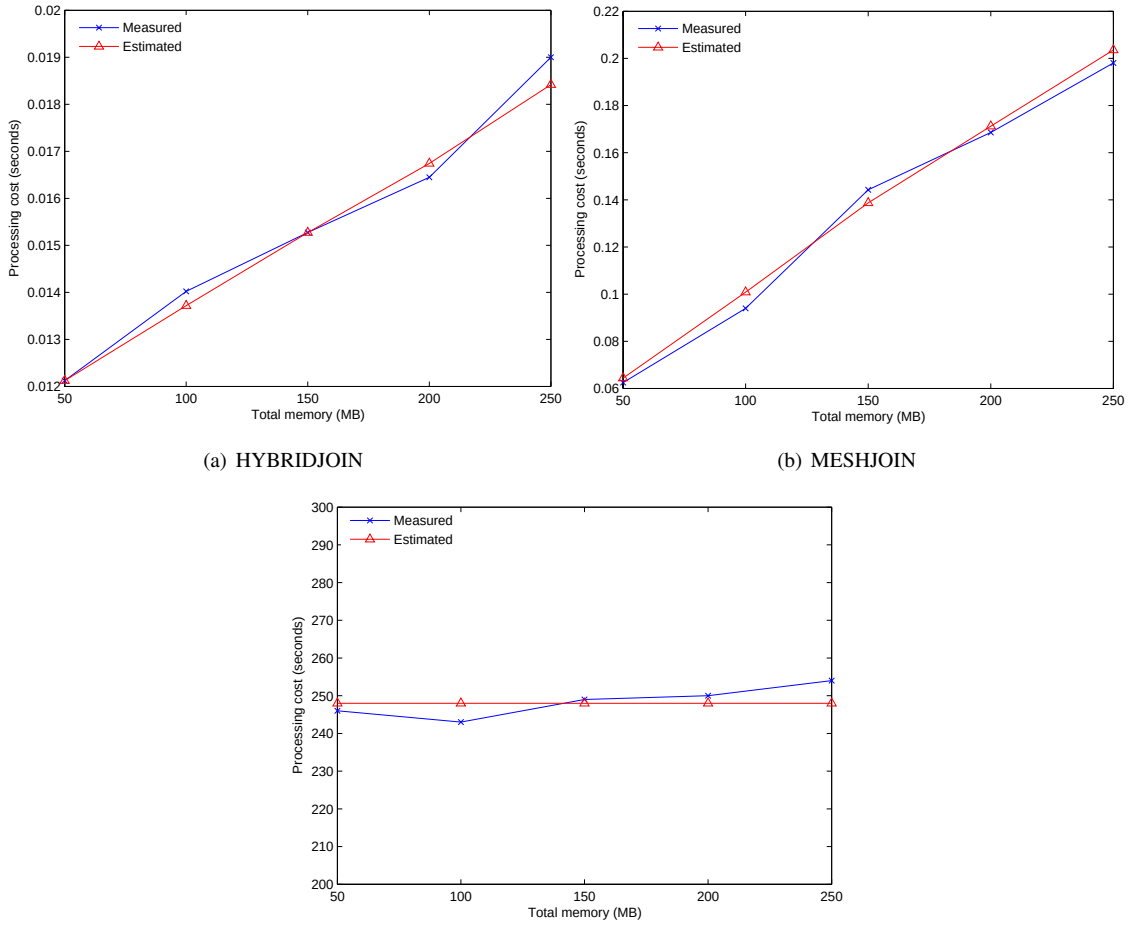
Figure 10: Experimental results

mance within the limited resources. We designed our own benchmark to test our approach according to current market economics. To validate our arguments we implemented a prototype of HYBRIDJOIN that demonstrates a significant improvement in service rate under limited memory. We also provide the open sources for our implementations.

In order to further improve the performance of HYBRIDJOIN, we will extend the implementation of the proposed join algorithm by dynamically ordering of disk based relation with respect to access frequency.

Source URL: The source of our implementations for HYBRIDJOIN, MESHJOIN and INLJ can be downloaded using given *URL*.

http://www.cs.auckland.ac.nz/research/groups/serg/hybridjoin/



(a) HYBRIDJOIN

(b) MESHJOIN

(c) INLJ

Figure 11: Cost validation

Acknowledgments.

The research work is being funded with the collaboration of The University of Auckland, New Zealand and Higher Education Commission (HEC), Pakistan.

References

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

[2] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

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

[4] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB '2003: Proceedings of the 29th International Conference on Very Large Data Bases*, pages 500–511. VLDB Endowment, 2003.

[5] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18:3–18, 1995.

[6] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. *SIGMOD Rec.*, 28(2):299–310, 1999.

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

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

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

[10] W. J. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient resumption of interrupted warehouse loads. *SIGMOD Rec.*, 29(2):46–57, 2000.

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

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

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

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

[15] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N. Frantzell. Supporting streaming updates in an active data warehouse. *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, April 15-20, 2007, Istanbul, Turkey*, pages 476–485, April 2007.

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

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

[18] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

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

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

[21] X. Zhang and E. A. Rundensteiner. Integrating the maintenance and synchronization of data warehouses using a cooperative framework. *Inf. Syst.*, 27(4):219–243, 2002.

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