View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AUT Scholarly Commons

HYBRIDJOIN for Near-Real-Time Data Warehousing

M. Asif Naeem, The University of Auckland, New Zealand

Gillian Dobbie, The University of Auckland, New Zealand

Gerald Weber, The University of Auckland, New Zealand

ABSTRACT

An important component of near-real-time data warehouses is the near-real-time integration layer. One important element in near-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, in MESHJOIN the performance of the algorithm is inversely proportional to the size of disk-based relation. Also, MESHJOIN cannot deal with intermittent streams efficiently, because tuples could wait for an undetermined time, thus defying the near-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.

Keywords: Near-real-time; data warehousing; stream-based join; data transformation; performance and tuning

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 (Pedersen, 2009) (Golfarelli & Rizzi, 2009b) (Golfarelli & Rizzi, 2009a) (Vassiliadis, 2009). Most data warehouses have already switched from a full refresh (Gupta & Mumick, 1999) (Zhang & Rundensteiner, 2002) (Zhuge, García-Molina, Hammer, & Widom, 1995) to an incremental refresh policy (W. Labio & Garcia-Molina, 1996) (W. J. Labio, Wiener, Garcia-Molina, & Gorelik, 2000) (W. Labio, Yang, Cui, Garcia-Molina, & Widom, 2000). Furthermore, the batch-oriented, incremental refresh approach is moving towards a continuous, incremental refresh approach (Thiele, Fischer, & Lehner, 2007)

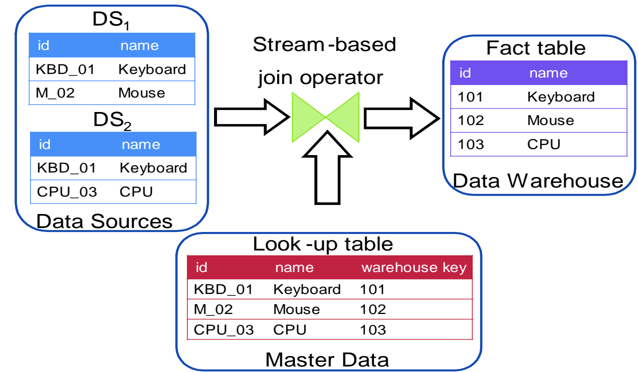
(Karakasidis, Vassiliadis, & Pitoura, 2005) (Tho Manh Nguyen, 2003).

With regards to terminology, data warehousing approaches that follow such a best-effort data freshness approach have various names. Frequently used terms are zero-latency, active, real-time or near-real-time data warehouses. The term near-real-time is the most descriptive in a context where there could be a confusion with real-time control systems, but for the sake of brevity, we will mostly use the term real-time in this paper where no such confusion is possible.

One important research area in the field of data warehousing is data transformation, since the updates coming from the data sources are often not in the format required for the data warehouse. For real-time data warehousing a continuous transformation from a source to target format is required, so the task becomes 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 updates with values from the master data. Enrichment in particular can often be expressed as a join between the update stream and the master data (Naeem, Dobbie, & Weber, 2008). One important example of enrichment is a key transformation. Normally the key used in the data source is different from that in the data warehouse and therefore needs to be replaced. 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. 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.

Figure 1: An example of stream-based join



In traditional data warehousing the update tuples are buffered in memory and joined when resources become available (Annita N. Wilschut & Apers, 1991) (Shapiro, 1986). Whereas, in real-time data warehousing these update tuples are joined immediately 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 (N. Polyzotis, Skiadopoulos, Vassiliadis, Simitsis, & Frantzell, 2007) (Neoklis Polyzotis, Skiadopoulos, Vassiliadis, Simitsis, & Frantzell, 2008) 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, we have identified two issues that have to be addressed. Firstly, the throughput of MESHJOIN is inversely proportional to the size of the disk-based master data table. Secondly, the algorithm cannot deal with an intermittent update stream efficiently. An intermittent stream is a stream that is dropping to a rate close to zero tuples per unit

of time for periods of time. A detailed explanation of these issues is provided in Section 3.

The Index Nested Loop Join (INLJ) (Ramakrishnan, 1999) is traditionally considered for non-stream data, but it can easily be set up so that it joins a continuous data stream with a disk-based relation, which is capable of dealing with intermittent data streams. However, every index has to be considered non-clustered with respect to the stream data. This is because stream data arrive in the order that the updates are performed. The natural assumption is e.g. that purchases are random. INLJ is known to be inefficient for non-clustered index access. The disk I/O cost cannot be amortized over multiple tuples of the stream and eventually produces a low service rate.

Based on these observations, we propose a 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 can reduce the disk I/O cost by guaranteeing that every partition 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 a disk partition 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 intermittent streams.

The rest of this 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 throughput for join algorithms. Some techniques have been introduced already to process join queries over continuous streaming data (Golab & Özsu, 2003) (Babu & Widom, 2001) (Hammad, Aref, & Elmagarmid, 2008) (Palma, Akbarinia, Pacitti, & Valduriez, 2009) (Kim & Park, 2005) (Nguyen, Brezany, Tjoa, & Weippl, 2005). 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) (Annita N. Wilschut & Apers, 1991) (A. N. Wilschut & Apers, 1990) 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) (Ives, Florescu, Friedman, Levy, & Weld, 1999) with a two stage join algorithm is an extension of SHJ. The XJoin algorithm (Urhan & Franklin, 2000) is another extension of SHJ. Hash-Merge Join (HMJ) (Mokbel, Lu, & Aref, 2004) which is also based on symmetric join algorithm, uses push technology and consists of two phases, hashing and merging.

Early Hash Join (EHJ) (Lawrence, 2005) 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) (N. Polyzotis, et al., 2007) (Neoklis Polyzotis, et al., 2008), 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 (Chakraborty & Singh, 2009) 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 in practice.

3. PRELIMINARIES: MESHJOIN

In this section we summarize the constraints on the MESHJOIN and INLJ algorithms. At the end of the section we outline 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 partitions. Once the last partition is read, it again starts from the first partition. The algorithm contains a buffer, called the disk buffer, to store each disk partition in memory one at a time. The algorithm uses a hash table to store the stream tuples, while the key attribute for each tuple is stored in the queue. All partitions in the queue are equal in size. The total number of partitions is equal to the number of partitions on the disk while the size of each partition on the disk is equal to the size of the disk buffer. There is a stream buffer of negligible size that is used to hold the fast stream if required.

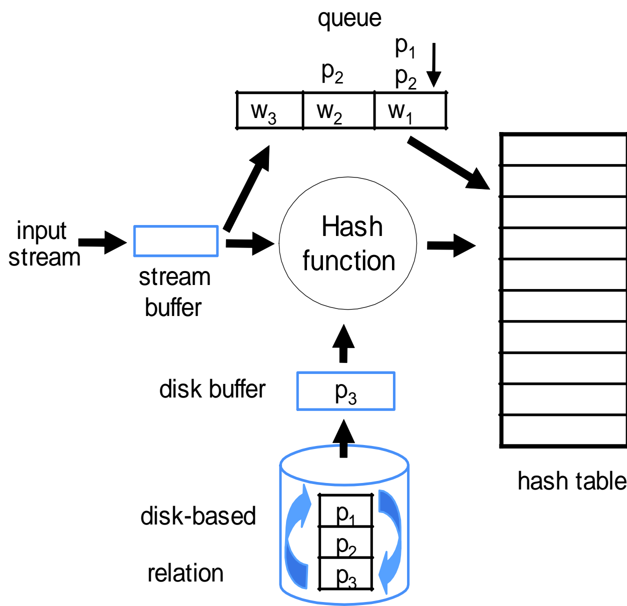
In each iteration the algorithm reads one disk partition into the disk buffer and loads a chunk of stream tuples into the hash table while also placing their key attributes in the queue. After loading the disk partition into memory it joins each tuple from that partition with matching stream tuples in the hash table. Before the next iteration the oldest stream tuples are removed from the hash table with their key attribute values from the queue. All chunks of the stream in the queue are advanced by one step. In the next iteration the algorithm replaces the current disk partition with the next one, loads a chunk of new stream tuples into the hash table and places their key attributes values in the queue, and repeats the above procedure.

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 the disk buffer b (i.e. $k=N_R/b$). This constraint ensures that a stream tuple that is loaded into memory is matched against the entire disk relation before it expires.

An overview of MESHJOIN is presented in Figure 2 where we consider only three partitions in the queue, with the same number of partitions on disk. At any time t, for example when disk partition p_3 is in memory the status of the stream tuples in memory can be explained. In the queue w_1 tuples have already joined with disk partition

p_1 and p_2 and therefore after joining with partition p_3 they will be dropped out of memory. While tuples w_2 have joined with partition p_2 only and therefore, after joining with partition p_3 they will advance one step in the queue. Finally, tuples w_3 have not joined with any disk partition and they will also advance one step in the queue after joining with partition p_3. Once the algorithm completes the cycle of R, it again starts loading sequentially from the first partition.

Figure 2: Example of MESHJOIN when disk partition p_3 is in memory

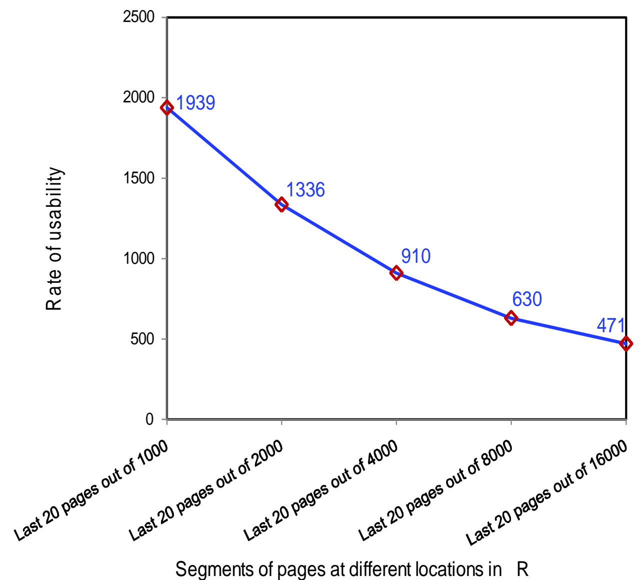


The MESHJOIN algorithm successfully amortizes the fast arrival rate of the incoming stream by executing the join of disk partitions with a large number of stream tuples. However there are still some further issues that exist in the algorithm. Firstly due to the sequential access of R, the algorithm reads the unused or less used partitions 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) = $\frac{1}{2}$ (*seek time* + *processing time*) for the whole of R

To determine the access rate of disk partitions of R we performed an experiment using a benchmark that is based on Zipf's Law to model commercial applications (Knuth, 1998) (Anderson, 2006), 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 pages at different locations of R. From the results shown in Figure 3 it can be seen 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.

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



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 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 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 bursty stream effectively.

4. HYBRIDJOIN

In previous section we highlighted observations related to the MESHJOIN and INLJ algorithms. As a solution to the stated problems we propose a stream-based join algorithm called Hybrid Join (HYBRIDJOIN). In HYBRIDJOIN we address two major aims which are not supported in MESHJOIN: (a) efficient access of disk-based relation R by loading only the useful part of R into memory, (b) dealing with bursty streams effectively. This section describes the data structures, pseudo-code and run time analysis of HYBRIDJOIN. We also present the cost model that is used for estimating the cost of our algorithm, and for tuning the algorithm.

4.1 Data structures and architecture

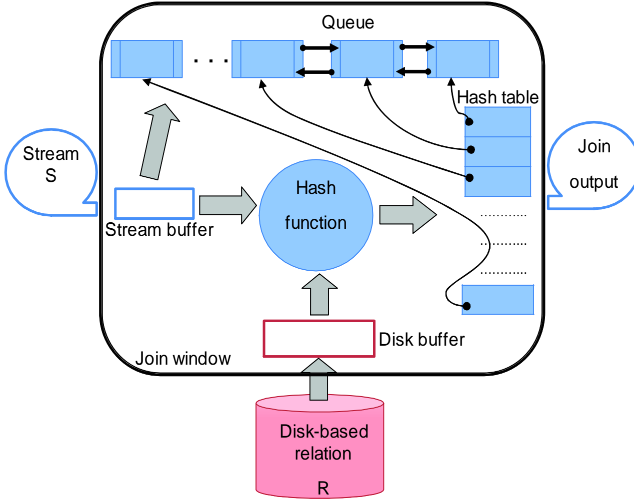
The data structures that HYBRIDJOIN uses are shown in Figure 4. Like in MESHJOIN key components of HYBRIDJOIN are disk buffer, hash table, queue and stream buffer. The disk-based relation R and stream S are the inputs. Contrary to MESHJOIN in HYBRIDJOIN we assume that R contains the unique values of join attribute and has an index on it. We also assume that the values of join attribute are sorted. The disk partition of size v_P from relation R is loaded into the disk buffer in memory. The queue is used to store the value of the join attribute and each node in the queue also contains the addresses of its one step neighbour nodes. Unlike the queue in MESHJOIN we implement an extra feature of random deletion in our HYBRIDJOIN queue by using a doubly-linked-list.

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 partition is loaded into memory using the join attribute value from the queue as an index, instead of only matching one tuple as in INLJ; the algorithm matches the disk partition with all the matching tuples in the queue. This helps to amortize the expensive disk I/O cost over 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 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.

Figure 4: Data structures used in 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. Initially since the hash table is empty, h_S is assigned to stream input size w where h_S is the total number of slots in the hash table H (line 1). The algorithm consists of two loops: one is called the outer loop while the other one is called the inner loop. The outer loop which is an endless loop is used to read the stream input into the hash table (line 2). While the inner loop is used to probe the hash table (line 9). In each outer loop iteration, the algorithm examines the availability of stream tuples in the stream buffer. If the required number of stream tuples available, the algorithm reads w tuples of the stream and loads them into the hash table while placing their join attribute values in the queue. Once the stream input is read the algorithm resets the value of w to 0 (line 3-6). The algorithm then reads the oldest value of a join attribute from the queue and loads a disk partition p into the disk buffer, using that join attribute value as an index (line 7, 8). After loading the disk partition into memory the inner loop starts and for each iteration of the inner loop the algorithm reads one disk tuple from the disk buffer and probes the hash table. In the case of a match, the algorithm generates the join output. Since the hash table is multi-hash-map, there can be more than one match against one disk tuple. After generating the join output the algorithm deletes all matched tuples from the hash table along with the corresponding nodes from the queue. Finally, the algorithm increments w with the number of vacated slots in the hash table (line 9-15).

Figure 5: Pseudo-code for HYBRIDJOIN

```
HYBRIDJOIN algorithm
Input: A disk based relation R with an index on join
attribute and a stream of updates S
Output: Stream R ⋈ S
Parameters: w tuples of S and a partition p of R
Method:
1. w ← h_S
2. While (true)
3.    If (available stream tuples ≥ w)
4.       Read w tuples from stream buffer and load
         them into H while adding their join attribute
         values in Q
5.       w ← 0
6.    EndIf
7.    Read the oldest join attribute value from Q
8.    Load a partition p of R into the disk buffer using
      that join attribute value as an index
9.    For each tuple r in partition p
10.      If r ∈ H
11.         Output r ⋈ H
12.         Delete all matched tuples from H and
            the corresponding nodes from Q
13.         w ← w + number of matching tuples found in H
14.      EndIf
15.   EndFor
16. EndWhile
```

4.3 Asymptotic runtime analysis

We compare the asymptotic runtime of HYBRIDJOIN with that of MESHJOIN and INLJ. As a unit of measurement we use the time needed

to process a stream chunk. The time needed to process a single tuple is the inverse of the service rate, which is the number of tuples processed in a time interval. The unit of measurement used here has the advantage, that "smaller is better" in accordance with common usage in asymptotic analysis of algorithms. Every stream section can be viewed as a binary sequence, and by viewing this binary sequence as a natural number, we can apply asymptotic complexity classes to functions on stream sections as binary numbers. Note therefore that the following theorems do not use functions on input lengths, but on binary numbers representing stream sections. We denote the time needed to process stream section s as MEJ(s) for MESHJOIN, as INLJ(s) for index nested loop join, and as HYJ(s) for HYBRIDJOIN. 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_S 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 partitions is as fast as sequential access (seek time is a constant factor). For MESHJOIN with its cyclic access pattern for R, every partition of R is accessed exactly once after every h_S stream tuples. We have to show that for HYBRIDJOIN no partition is accessed more frequently. For that we look at an arbitrary partition 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_S 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 $i+h_S$ 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 $i+h_S$, because all earlier stream tuples that match data on p have been deleted from the queue. This proves the theorem.

Comparison with INLJ:

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

Proof: INLJ performs a constant number of disk accesses per stream tuple. For the theorem it suffices to prove that HYBRIDJOIN performs 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.

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

4.4 Cost model

In this section we derive the general formulas to calculate the cost of our proposed HYBRIDJOIN. Since it is important to compare our cost model with the cost model presented for MESH-

JOIN in (Neoklis Polyzotis, et al., 2008) we use the same notation where possible and also calculate the cost in terms of memory and processing time. Equation (1) describes the total memory used to implement the algorithm (excluding the stream buffer). Equation (3) calculates the processing cost for w tuples while the average size for w can be calculated using Equation (2). The service rate can be calculated using Equation (4). The symbols used in the equations are specified in Table 1.

4.4.1 Memory cost

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

$$M = v_P + \alpha(M - v_P) + (1 - \alpha)(M - v_P) \quad (1)$$

We are not including the memory reserved for the stream buffer because it is negligible (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_S
- Size of disk buffer d
- Size of disk-based relation R_t
- The exponent value for benchmark e

Table 1: Notations used for cost estimation of HYBRIDJOIN

Parameter name	**Symbol**
Total allocated memory (bytes)	M
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)	v_P
Size of disk tuple (bytes)	v_R
Size of disk buffer (tuples)	d
Memory weight for hash table	α
Memory weight for queue	$(1- \alpha)$
Size of hash table (tuples)	h_S
Size of disk-based relation R (tuples)	R_t
Exponent value for benchmark	e
Cost to read one disk partition into disk buffer (nanosecs)	$c_{I/O}(v_P)$
Cost to probe one disk tuple into hash table (nanosecs)	c_H
Cost to generate output for one tuple (nano sec)	c_O
Cost to delete one tuple from hash and queue (nanosecs)	c_E
Cost to read one tuples into stream buffer (nanosecs)	c_S
Cost to add one tuples into hash and queue (nanosecs)	c_A
Cost for one loop iteration of HYBRIDJOIN (sec)	c_{loop}

In our experiments w is directly proportional to the size of the hash table h_S and the size of the disk buffer d, and is inversely proportional to the size of disk-based relation R_t. The fourth parameter represents the exponent value for Zipfian distribution, explained in Section 5, and by using an exponent value of 1 we approximately model the 80/20 Rule (Anderson, 2006) for market sales. Therefore, the formula for w is:

$$w \propto \frac{h_S \times d \times e}{R_t}$$

$$w = k \frac{h_S \times d}{R_t} \quad (2)$$

where k is a constant influenced by system parameters. We obtained the value of k from measurements, in our setup it is is 1.36.

On the basis of w we can calculate the processing cost for one step of the iteration. In order to calculate the cost for one loop iteration the major components are:

Cost to read one disk partition = $c_{I/O}(v_P)$.

Cost to probe one disk partition into the hash table = $\frac{v_P}{v_R} c_H$.

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

Cost to delete w tuples from the hash table and the queue = $w.c_E$.

Cost to read w tuples from stream $S = w.c_S$.

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

By aggregation, the total cost for one loop iteration is:

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

Since in 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.

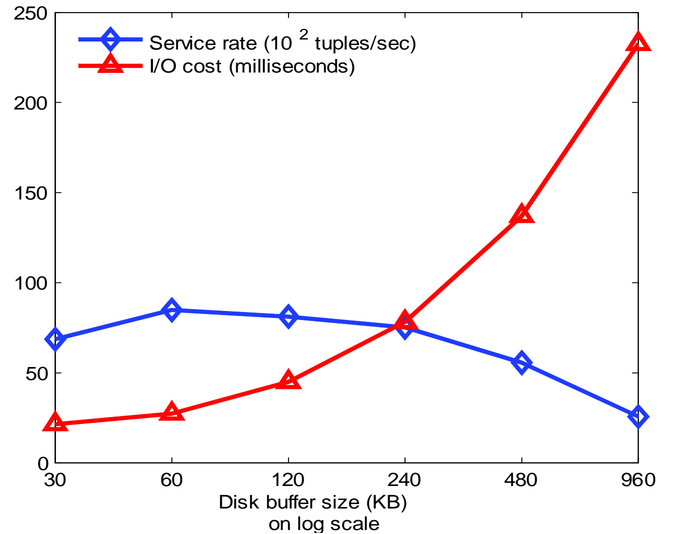
$$\mu = \frac{w}{c_{loop}} \qquad (4)$$

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} which is required to process these w tuples. In HYBRIDJOIN for a particular setting (M = 50MB) assuming the size of R and the exponent value for Zipfian distribution are fixed (R_t = 2 million 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 is 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 shows the relationship between the I/O cost and service rate as measured in experiments. From Figure 6 it can be observed that in the beginning, for a small disk buffer size, the service rate is also small because there are fewer matching tuples in the queue. However, the service rate increases with an increase in the size of the disk buffer due to 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 increase in the disk I/O cost and the decrease in memory size for the hash table.

Figure 6: Tuning of disk buffer



5. TESTS WITH LOCALITY OF DISK ACCESS

Crucial to the HYBRIDJOIN performance is the distribution of master data keys in the stream. If the distribution is uniform, then HYBRIDJOIN may perform worse than MESHJOIN, but by a

constant factor, in line with the theoretical analysis. Note however, that HYBRIDJOIN still has the advantage of being efficient for intermittent streams, while the original MESHJOIN would pause with intermittent streams, and leave tuples unprocessed for an open-ended period.

It is also obvious that HYBRIDJOIN has advantages if *R* contains unmatched 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.

Common non-uniform distributions are Zipfian distributions, which exhibit a power law similar to Zipf's law (Knuth, 1998). Zipfian distributions are discussed as a plausible model for sales (Anderson, 2006), where some products are sold frequently while most are sold rarely. The distribution generated using Zipf's law with an exponent 1 is close to the 80/20 Rule (Anderson, 2006) i.e. 80% of the sales are from 20% of the products.

We designed a generator for synthetic data that follows a Zipfian distribution. The generated benchmark is based on two characteristics: (a) the frequency of selling each product and it approximately models the 80/20 Rule, (b) the flow of sales transactions and it is a self-similar. By using the generated benchmark we demonstrate that HYBRIDJOIN performance improves when locality is considered, 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.

6. EXPERIMENTS

We performed an experimental evaluation of HYBRIDJOIN, using synthetic datasets. In this section we describe the environment of our experiments and analyze the results that we obtained using different scenarios.

6.1 Experimental setup

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.

Hardware specifications: We carried out our experimentation on a Pentium-IV 2X2.13GHz machine with 3G main and 160G disk memory under WindowsXP. We implemented the experiment in Java using the Eclipse IDE 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* is stored on disk using MySQL 5.0 database, while the bursty type of stream data is generated at run time using our own generator. Both the algorithms read master data from the database. In transformation, join is normally performed between the primary key of the lookup table and the foreign key in the stream tuple and therefore our HYBRIDJOIN supports join for one-to-many relationships and it can be extended for many-to-many relationships easily. 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.

We compare the performance of HYBRIDJOIN with MESHJOIN and INLJ while varying total allocated memory *M*, the size of relation *R* on disk, the value of the exponent for the Zipfian distribution, and the stream arrival rate λ. The other parameters such as the size of disk buffer, size of stream buffer, size of each disk tuple, size of each stream tuple, and the size of each node in the queue are considered fixed. The stream dataset we used to evaluate HYBRIDJOIN is based on Zipf's law and has two important characteristics, bursty and self-similarity (for details, see the appendix). We test the performance of all the algorithms by varying the Zipfian exponent value from 0 to 1.

Table 2: Data specifications

Parameter	Value
Memory	
Total allocated memory *M*	50MB to 250MB
Exponent	
Zipfian exponent value	0 to 1
Disk-based data	
Size of disk-based relation *R*	0.5 million to 8 million tuples
Size of each disk tuple	120 bytes
Stream data	
Size each stream tuple	20 bytes
Size of each nodes in queue	12 bytes
Stream arrival rate λ	125 to 2000 tuples/sec
Benchmark	
Based on	Zipf's law
Characteristics	Bursty and self-similar

Measurement strategy: The performance or service rate of the join is measured by calculating the maximum 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 it is assumed that no other application is running in parallel.

6.2 Experimental results

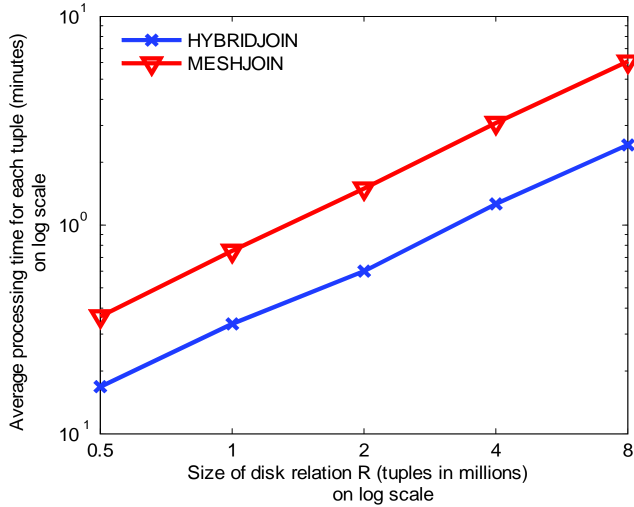
We conducted two kinds of evaluation. In Section 6.2.1 we compare the performance of all three approaches, while in Section 6.2.2 we validate the cost by comparing it with the predicted cost.

6.2.1 Performance comparison

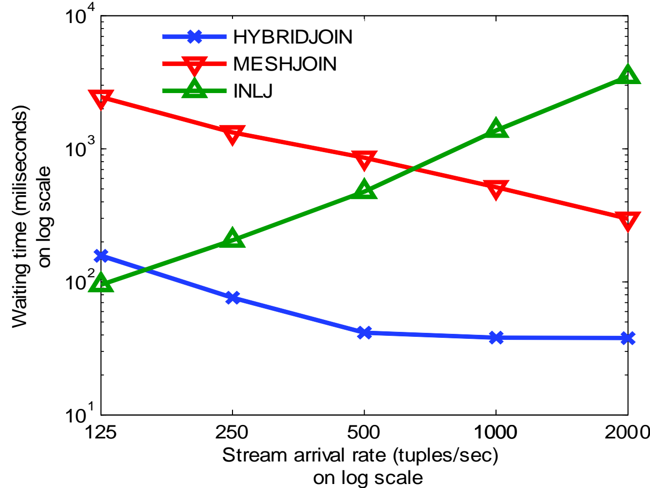
As the source code for MESHJOIN is not openly available, we implemented the MESHJOIN algorithm ourselves. In our experiments we compare the performance in two different ways. First, we compare HYBRIDJOIN with MESHJOIN with respect to the time, both *processing time* and *waiting time*. Second, we compare 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 7(a), presents the comparisons with respect to *processing time* on a log scale, while Figure 7(b) depicts the comparisons with respect to the *waiting time*. The terms *processing time* and *waiting time* have already been defined in Section 3. According to Figure 7(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 partitions with the same frequency without considering the rate of use of each partition on the disk. In HYBRIDJOIN an index based approach is implemented that never reads unused disk partitions of *R*. In this experiment we do not reflect the *processing time* for INLJ because it is constant even when the size of *R* changes.

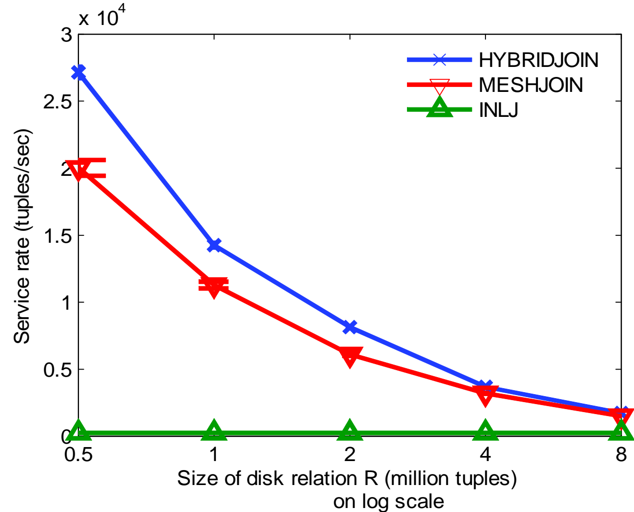
Figure 7: Performance evaluation



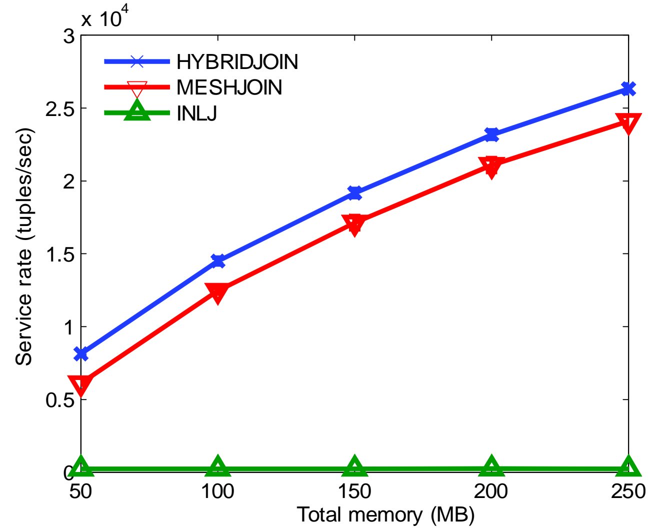
(a) Processing time



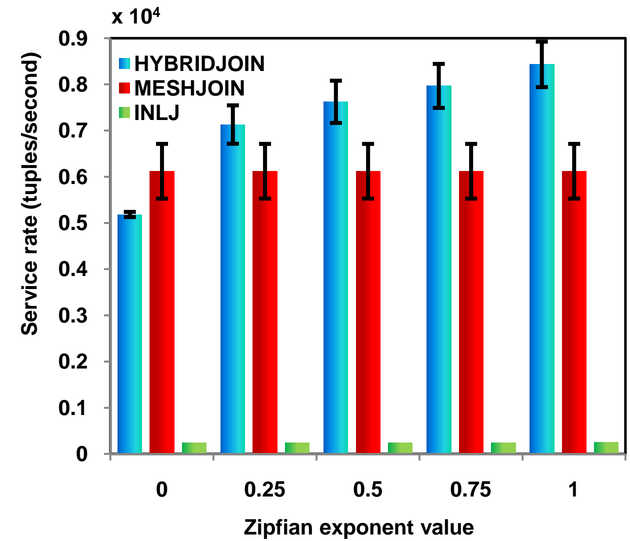
(b) Waiting time in case of low stream arrival rate



(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



(e) Performance comparison for varying Zipfian exponent

In the experiment shown in Figure 7(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 a faster incoming stream rate than the processing rate. Also this delay (*waiting time*) increases with an increase in the stream arrival rate.

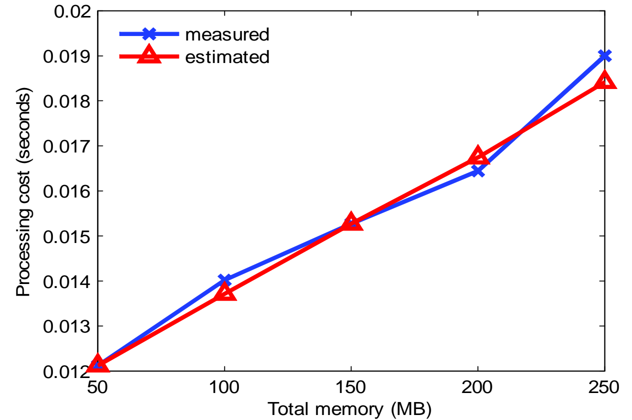
In the other two approaches the *waiting time* in MESHJOIN is greater than in HYBRIDJOIN. In HYBRIDJOIN since there is no constraint to

match each stream tuple with the whole of R, each disk invocation is not synchronized 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.

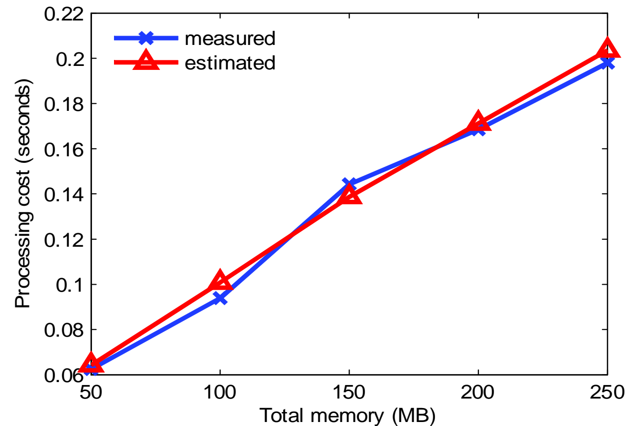
Performance comparisons with respect to service rate: In this category of our experiments we compare the performance of HYBRIDJOIN in terms of the service rate with the other two join algorithms by varying different parameters such as the total memory budget, the size of R, and the value of Zipfian exponent. In the experiment shown in Figure 7(c) we assume the total allocated memory for the join is fixed while the size of R varies exponentially. From Figure 7(c) it can be observed that for all sizes of R, the performance of HYBRIDJOIN is significantly better compared with the other join approaches. 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 7(d) depicts the comparisons of all three approaches. From Figure 7(d) it is clear that for all memory budgets the performance of HYBRIDJOIN is better as compared to the other two algorithms.

Finally, we evaluate the performance of HYBRIDJOIN by varying the skew in input stream S. To vary the skew, we vary the value of the Zipfian exponent e. In our experiments we allow it to range from 0 to 1. At 0 the input stream S is uniform and the skew increases as e increases. Figure 7(e) presents the results of our experiment. It is clear from Figure 7(e) that under all values of e except 0, HYBRIDJOIN performs considerably better than MESHJOIN and INLJ. Also this improvement increases with an increase in e. The plausible reason for this better performance in the case of HYBRIDJOIN is that the algorithm does not read unused parts of R into memory and it saves unnecessary I/O cost. Moreover, when e increases the input stream S gets more skewed and consequently, the I/O cost decreases due to an increase in the size of the unused part of R. However, when e is equal to 0, HYBRIDJOIN performs worse than MESHJOIN but it is only worse by a constant factor.

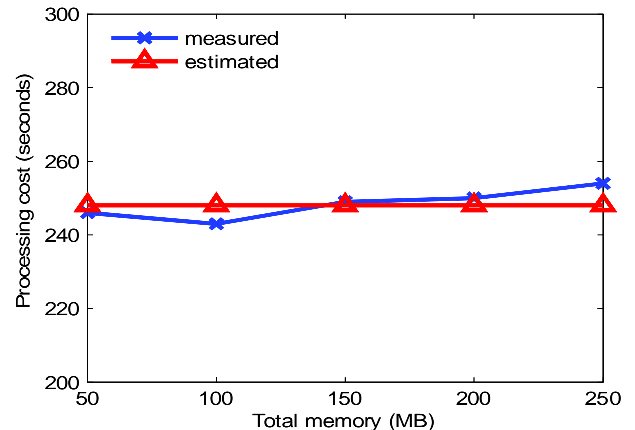
Figure 8: Cost validation



(a) HYBRIDJOIN



(b) MESHJOIN



(c) INLJ

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 8 presents the comparisons of both costs. In Figure 8 it is demonstrated that the predicted cost closely resembles the measured cost in every approach which validates the accuracy of our cost model.

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 bursty data streams. We developed a cost model and tuning methodology in order to achieve the maximum performance 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 implementations.

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

Source URL: The source of our implementations for HYBRIDJOIN, MESHJOIN and INLJ can be downloaded from:
http://www.cs.auckland.ac.nz/research/groups/serg/hybridjoin/

8. REFERENCES

Anderson, C. (2006). *The Long Tail: Why the Future of Business Is Selling Less of More*: Hyperion, pp. 130--135.

Babu, S., & Widom, J. (2001). Continuous queries over data streams. *SIGMOD Rec., 30*(3), 109-120.

Chakraborty, A., & Singh, A. (2009). *A partition-based approach to support streaming updates over persistent data in an active datawarehouse*. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing. Washington, DC, USA, 1-11.

Golab, L., & Özsu, M. T. (2003). *Processing sliding window multi-joins in continuous queries over data streams*. In Proceedings of the 29th International Conference on Very Large Data Bases. Berlin, Germany, 500-511.

Golfarelli, M., & Rizzi, S. (2009a). A Survey on Temporal Data Warehousing. *International Journal of Data Warehousing and Mining (IJDWM), 5*(1), pp. 1-17.

Golfarelli, M., & Rizzi, S. (2009b). What-if Simulation Modeling in Business Intelligence. *International Journal of Data Warehousing and Mining (IJDWM), 5*(4), pp. 24-43.

Gupta, A., & Mumick, I. S. (1999). *Maintenance of materialized views: problems, techniques, and applications* (Vol. 18). Cambridge, MA, USA: MIT Press.

Hammad, M. A., Aref, W. G., & Elmagarmid, A. K. (2008). Query processing of multi-way stream window joins. *The VLDB Journal, 17*(3), 469-488.

Ives, Z. G., Florescu, D., Friedman, M., Levy, A., & Weld, D. S. (1999). An adaptive query execution system for data integration. *SIGMOD Rec., 28*(2), 299-310.

Karakasidis, A., Vassiliadis, P., & Pitoura, E. (2005). *ETL queues for active data warehousing*. In Proceedings of the 2nd

International Workshop on Information Quality in Information Systems. Baltimore, Maryland, 28-39.
Kim, J., & Park, S. (2005). Periodic Streaming Data Reduction Using Flexible Adjustment of Time Section Size. *International Journal of Data Warehousing and Mining (IJDWM), 1*(1), pp. 37-56.
Knuth, D. E. (1998). *The Art of Computer Programming* (Vol. 3): Addison-Wesley Longman Publishing Co., pp. 400-401.
Labio, W., & Garcia-Molina, H. (1996). *Efficient Snapshot Differential Algorithms for Data Warehousing*. In Proceedings of the 22th International Conference on Very Large Data Bases. San Francisco, CA, USA, 63-74.
Labio, W., Yang, J., Cui, Y., Garcia-Molina, H., & Widom, J. (2000). *Performance Issues in Incremental Warehouse Maintenance*. In Proceedings of the 26th International Conference on Very Large Data Bases. San Francisco, CA, USA, 461-472.
Labio, W. J., Wiener, J. L., Garcia-Molina, H., & Gorelik, V. (2000). Efficient resumption of interrupted warehouse loads. *SIGMOD Rec., 29*(2), 46-57.
Lawrence, R. (2005). *Early Hash Join: a configurable algorithm for the efficient and early production of join results*. In Proceedings of the 31st International Conference on Very Large Data Bases. Trondheim, Norway, 841-852.
Mokbel, M. F., Lu, M., & Aref, W. G. (2004). *Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results*. In Proceedings of the 20th International Conference on Data Engineering, ICDE 2004. Washington, DC, USA, 251-263.
Naeem, M. A., Dobbie, G., & Weber, G. (2008). *An Event-Based Near Real-Time Data Integration Architecture*. In Proceedings of the 12th Enterprise Distributed Object Computing Conference Workshops, EDOCW '08. Washington, DC, USA, 401-404.
Nguyen, T. M., Brezany, P., Tjoa, A. M., & Weippl, E. (2005). Toward a Grid-Based Zero-Latency Data Warehousing Implementation for Continuous Data Streams Processing. *International Journal of Data Warehousing and Mining (IJDWM), 1*(4), pp. 22-55.
Palma, W., Akbarinia, R., Pacitti, E., & Valduriez, P. (2009). DHTJoin: processing continuous join queries using DHT networks. *Distrib. Parallel Databases, 26*(2-3), 291-317.
Pedersen, C. T. a. T. B. (2009). A Survey of Open Source Tools for Business Intelligence. *International Journal of Data Warehousing and Mining (IJDWM), 5*(3), pp. 56-75.
Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., & Frantzell, N. (2008). Meshing Streaming Updates with Persistent Data in an Active Data Warehouse. *IEEE Trans. on Knowl. and Data Eng., 20*(7), 976-991.
Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., & Frantzell, N. E. (2007). *Supporting Streaming Updates in an Active Data Warehouse.* In Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, Istanbul, Turkey, 476-485.
Ramakrishnan, R. (1999). *Database Management Systems* (2nd ed.): McGraw-Hill, Inc., pp. 337-339.
Shapiro, L. D. (1986). Join processing in database systems with large main memories. *ACM Trans. Database Syst., 11*(3), 239-264.
Thiele, M., Fischer, U., & Lehner, W. (2007). *Partition-based workload scheduling in living data warehouse environments*. In Proceedings of the ACM tenth international workshop on Data warehousing and OLAP. Lisbon, Portugal, 57-64.

Tho Manh Nguyen, A. M. T. (2003). *Zero-latency data warehousing for heterogeneous data sources and continuous data streams.* In Proceedings of the iiWAS'2003 - The Fifth International Conference on Information Integrationand Web-based Applications Services, Jakarta, Indonesia, 55-64.

Urhan, T., & Franklin, M. (2000). XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin, 23*(27).

Vassiliadis, P. (2009). A Survey of Extract–Transform–Load Technology. *International Journal of Data Warehousing and Mining (IJDWM), 5*(3), 1-27.

Wilschut, A. N., & Apers, P. M. G. (1990). *Pipelining in query execution.* In Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications,. PARBASE-90, Miami Beach, FL, USA, 562.

Wilschut, A. N., & Apers, P. M. G. (1991). *Dataflow query execution in a parallel main-memory environment*. In Proceedings of the first International Conference on Parallel and Distributed Information Systems. Miami, Florida, United States, 68-77.

Zhang, X., & Rundensteiner, E. A. (2002). Integrating the maintenance and synchronization of data warehouses using a cooperative framework. *Information Systems, 27*(4), 219-243.

Zhuge, Y., García-Molina, H., Hammer, J., & Widom, J. (1995). *View maintenance in a warehousing environment*. In Proceedings of the 1995 ACM SIGMOD international conference on Management of data. San Jose, California, United States, 316-327.

APPENDIX

BENCHMARK

In order to demonstrate the behavior of the algorithm with a bursty stream, 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 benchmark is shown in Figure 9. In Figure 9 STREAMGENERATOR is the main procedure while GETDISTRIBUTIONVALUE and SWAPSTATUS 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 GETDISTRIBUTIONVALUE 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 SWAPSTATUS procedure (line 11 to 14). The SWAPSTATUS 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 GETDISTRIBUTIONVALUE line (15 to 17). For each call to procedure GETDISTRIBUTIONVALUE, it returns the random value by implementing Zipf's law with exponent value equal to 1 (line 28 to 31).

Procedure *STREAMGENERATOR*

1. *totalCurrentBandwidth* ← 0
2. *timeInChosenUnit* ← o
3. *on* ← *false*
4. *d* ← *GETDISTRIBUTIONVALUE( )*
5. **For** *i=1* to *N*
6. *PriorityQueue.enqueue(d,bandwidth= Math.power(2,i),timeInChosenUnit=currentTime( ))*
7. **EndFor**
8. *current* ← *PriorityQueue.dequeue( )*
9. **While** (*true*)
10. Wait(*oneStep*)
11. **If** (*currentTime( )* > *current.timeInChosenUnit*)
12. *current* ← *PriorityQueue.dequeue( )*
13. *SWAPSTATUS(current)*
14. **EndIf**
15. **For** *j=1* to *totalCurrentBandwidth*
16. **Output** *GETDISTRIBUTIONVALUE( )*
17. **EndFor**
18. **EndWhile**

Procedure *SWAPSTATUS (current)*

19. *timeInChosenUnit* ← (*current.timeInChosenUnit* + *getNextRandom()* × *oneStep* × *current.bandwidth*)
20. **If** (*on*)
21. *totalCurrentBandwidth* ← *totalCurrentBandwidth* – *current.bandwidth*
22. *on* ← *false*
23. **Else**
24. *totalCurrentBandwidth* ← *totalCurrentBandwidth* + *current.bandwidth*
25. *on* ← *true*
26. **EndIf**
27. *PriorityQueue.enqueue(current)*

Procedure *GETDISTRIBUTIONVALUE*

28. *sumOfFrequency* ← $\int \frac{1}{x} dx_{atx-max} - \int \frac{1}{x} dx_{atx-min}$
29. *random* ← *getNextRandom()*
30. *distributionValue* ← *inverseIntegralOf* (*random* x *sumOfFrequency* + $\int \frac{1}{x} dx_{atx-min}$)
31. **return** $\lfloor distributionValue \rfloor$

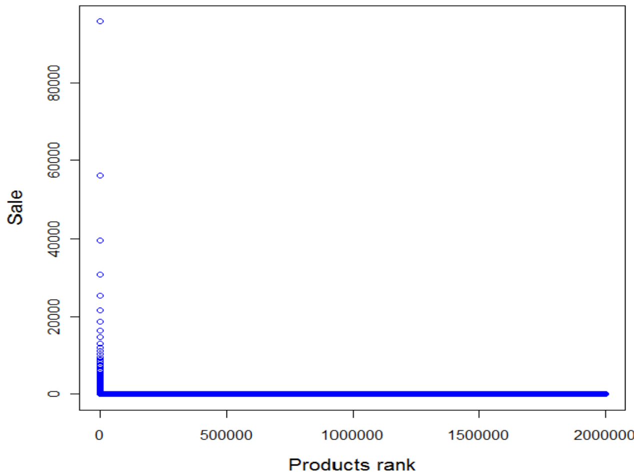
Figure 9: Pseudo-code for benchmark

The experimental representation of our benchmark is shown in Figure 10 and Figure 11, 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

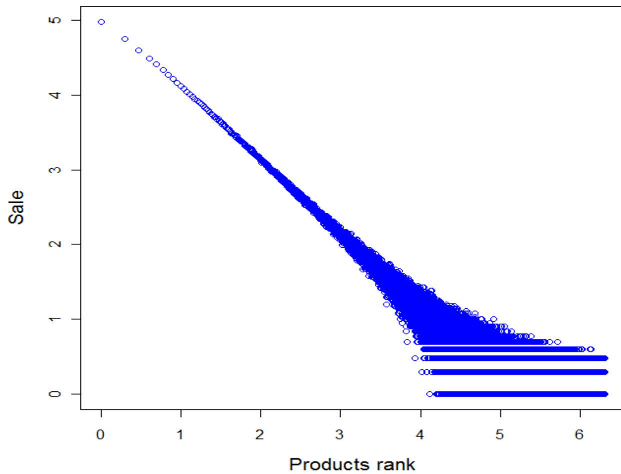
of selling each product while the other is the flow of these sales transactions. Figure 10 validates the first characteristic about real market sales. In Figure 10 the x-axis represents the variety of products while the y-axis represents the sales. Therefore, from Figure 10 it can be observed that only 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 data in which only a small portion of *R* is accessed again and again while the rest of *R* is accessed rarely.

Figure 10: A skewed distribution based on Zipf's law using exponent value is equal to1



(a) on plain scale



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

Figure 11: An input stream having bursty and self-similarity type of characteristics

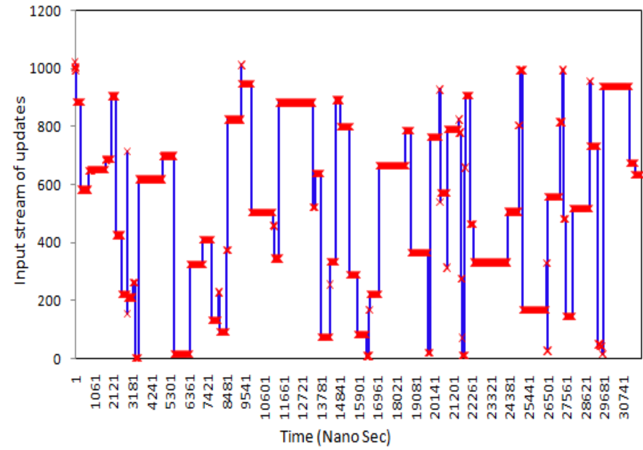


Figure 11 represents the flow of transactions, which is the second characteristic of our benchmark. From Figure 11 it is clear that the flow of transactions varies with time and is bursty rather than appearing at a regular rate.