Knowl Inf Syst (2014) 40:615–637
DOI 10.1007/s10115-013-0653-7

REGULAR PAPER

Efficient processing of streaming updates with archived master data in near-real-time data warehousing

M. Asif Naeem · Gillian Dobbie · Gerald Weber

Received: 20 November 2011 / Revised: 15 February 2013 / Accepted: 14 April 2013 /
Published online: 5 May 2013
© Springer-Verlag London 2013

Abstract In order to make timely and effective decisions, businesses need the latest information from data warehouse repositories. To keep these repositories up-to-date with respect to end user updates, near-real-time data integration is required. An important phase in near-real-time data integration is data transformation where the stream of updates is joined with disk-based master data. The stream-based algorithm MESHJOIN (Mesh Join) has been proposed to amortize disk access over fast streams. MESHJOIN makes no assumptions about the data distribution. In real-world applications, however, skewed distributions can be found, such as a stream of products sold, where certain products are sold more frequently than the remainder of the products. The question arises is how much does MESHJOIN lose in terms of performance by not adapting to data skew. In this paper we perform a rigorous experimental study analyzing the possible performance improvements while considering typical data distributions. For this purpose we design an algorithm Extended Hybrid Join (X-HYBRIDJOIN) that is complementary to MESHJOIN in that it can adapt to data skew and stores parts of the master data in memory permanently, reducing the disk access overhead significantly. We compare the performance of X-HYBRIDJOIN against the performance of MESHJOIN. We take several precautions to make sure the comparison is adequate and focuses on the utilization of data skew. The experiments show that considering data skew offers substantial room for performance gains that cannot be found in non-adaptive approaches such as MESHJOIN. We also present a cost model for X-HYBRIDJOIN, and based on that cost model, the algorithm is tuned.

M. A. Naeem (✉)
School of Computing and Mathematical Sciences, Auckland University of Technology, Private Bag 92006, Auckland, New Zealand
e-mail: mnae006@aucklanduni.ac.nz; mnaeem@aut.ac.nz

G. Dobbie · G. Weber
Department of Computer Science, The University of Auckland, Private Bag 92019, Auckland, New Zealand
e-mail: gill@cs.auckland.ac.nz

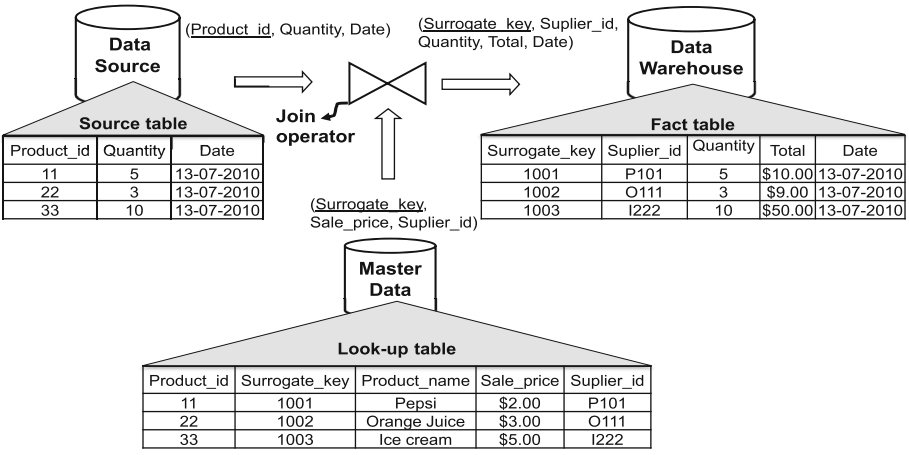
G. Weber
e-mail: gerald@cs.auckland.ac.nz

Keywords Near-real-time data warehousing · Stream-based join · Data transformation · Performance and tuning

1 Introduction

Near-real-time data warehouse deployments are driving an evolution to more aggressive data freshness levels. The tools and techniques for delivering these new service levels are evolving rapidly [10,14,22]. In the beginning, most data warehouses refreshed all content fully during each load cycle. However, due to rapid growth in the size of warehouses and the increasing demand of information freshness, it became infeasible to meet business needs. Therefore, the data acquisition mechanism in warehouses was changed from full refresh to an incremental refresh strategy, in which new data are added to the warehouse without requiring a complete reload [16–18]. Although this strategy is more efficient than the traditional one, it is still batch-oriented as a fraction of the data is propagated toward the warehouse after a particular timestamp. In order to overcome update delays, these batch-oriented and incremental refresh strategies are being replaced with a continuous refresh strategy [4,7–9,25,34,35]; that is, data are being captured and propagated to the data warehouse in near-real-time fashion in order to support high levels of data freshness.

An important operation in data integration is the transformation of the source data to a required format. Content enrichment is an example of such a transformation. In content enrichment master data attributes are added to source data [12]. A special case of content enrichment is the replacement of a *source data key* with a *warehouse key*. We consider an example of an inventory sales system, as shown in Fig. 1. The data source is shown as a source table, and it contains attributes *product_id*, *quantity* and *date*. The look-up table that stores master data contains attributes *product_id*, *surrogate_key*, *product_name*, *sale_price* and *supplier_id* with an index on attribute *product_id*. Let us assume that before loading the source table records into the data warehouse, they need to be enriched with certain information from the look-up table. In our example, the additional attributes are *supplier_id* and *total*, and the *surrogate_key* replaces the *product_id*. A join operator is required to perform this



Product_id	Quantity	Date
11	5	13-07-2010
22	3	13-07-2010
33	10	13-07-2010

Surrogate_key	Suplier_id	Quantity	Total	Date
1001	P101	5	$10.00	13-07-2010
1002	O111	3	$9.00	13-07-2010
1003	I222	10	$50.00	13-07-2010

Product_id	Surrogate_key	Product_name	Sale_price	Suplier_id
11	1001	Pepsi	$2.00	P101
22	1002	Orange Juice	$3.00	O111
33	1003	Ice cream	$5.00	I222

Fig. 1 An example of stream-based join

enrichment task. In the context of near-real-time data warehousing, one of the significant factors for choosing the join operator is that both the inputs for the join come from different sources and arrive at different rates. The source data are not a table but a high volume stream, and it has a bursty nature while the look-up table is disk-based. The access rate of the look-up table is comparatively slow due to disk I/O cost; therefore, a bottleneck is created during the join execution. The challenge in this case is to eliminate this bottleneck by amortizing disk I/O cost over a fast stream of updates. An alternative approach would be to try to put the whole disk-based relation into memory. In some cases this alternative can be feasible. But a number of scenarios exist where this alternative is not applicable, e.g., if the join is to be performed on a single computer where the bulk of memory is used for other purposes. Similarly, for intermittent streams, a main memory approach would keep the memory occupied even when no stream data are arriving. In the limited-memory approaches here, in contrast there is no such waste of resources.

A novel algorithm Mesh Join (MESHJOIN) [26,27] has been designed especially for joining a continuous stream with a disk-based relation, such as the scenario in active data warehouses. The algorithm makes no assumptions about data distribution or the organization of the master data. Experiments using MESHJOIN have shown that the algorithm performs worse with skewed data. The MESHJOIN algorithm is a hash join, where the stream serves as the build input and the disk-based relation serves as the probe input. The algorithm performs a staggered execution of the hash table build in order to load in stream tuples more steadily. However, there are some issues that plague MESHJOIN, such as suboptimal distribution of memory among the join components and an inefficient strategy for accessing the disk-based relation [23]. Although the MESHJOIN algorithm efficiently amortizes the disk I/O cost over fast input streams, the question remains how much potential for improvement remains untapped due to the algorithm not being able to adapt to common characteristics of real-world applications. In this paper we focus on one of the most common characteristics, a skewed distribution. Such distributions arise in practice, for example, current economic models show that in many markets select few products are bought with higher frequency [2]. Therefore, in the input stream, the sales transactions related to those products are the most frequent. In MESHJOIN, the algorithm does not consider the frequency of stream tuples and does not need an index structure on the master data. This can be useful in some circumstances, but still in many other cases one obviously wants to use an index to gain maximum performance. We propose an adaptive algorithm called Extended Hybrid Join (X-HYBRIDJOIN). The key feature of X-HYBRIDJOIN is that the algorithm stores the mostly used portion of the disk-based relation, which matches the frequent items in the stream, in memory. As a result, this reduces the I/O cost substantially, which improves the performance of the algorithm.

There are two major differences between X-HYBRIDJOIN and MESHJOIN. Firstly, the hash join component is modified so that it can make use of an index. Secondly, X-HYBRIDJOIN caches frequently used master data. Since we want to compare MESHJOIN and X-HYBRIDJOIN, it is important to clarify which change leads to the performance improvement. Therefore, we also present an intermediate step, HYBRIDJOIN, which implements only the first modification, and we compare all three algorithms. For X-HYBRIDJOIN we assume a scenario, where the master data itself changes infrequently, and likewise, the skewed frequency distribution changes only infrequently. Based on this, we assume that it is sufficient to reset the algorithms at these times, which also resets the non-swappable part of the disk buffer. If the frequency distribution changes only infrequently, it is feasible to estimate the frequency of tuples, e.g., by analyzing the stream in a sample time interval. The data can be sorted according to the estimated frequency. In principle

it would be desirable to overcome this requirement, and we will address this in future work.

Since our purpose is primarily to gauge performance with skewed distributions, we consider a very clean, artificial dataset that exactly exhibits a well-understood type of skew, a power law, with respect to the frequency that master data tuples are referenced in stream tuples. In all our experiments, the master data are sorted with respect to access frequency.

The algorithm has some limitations with respect to the frequency distribution. If the distribution is completely uniform, X-HYBRIDJOIN performs worse than MESHJOIN. The difference is a constant factor, as shown in an experiment in Sect. 5, Fig. 3c. Another limitation is the explicitly stated fact that the algorithm is designed for slowly changing master data and the assumption that the frequency also changes slowly.

The remainder of the paper is structured as follows. A review of the related work is presented in Sect. 2. Section 3 describes an intermediate modification, HYBRIDJOIN, while Sect. 4 presents X-HYBRIDJOIN with its algorithm and cost model. In Sect. 5, we compare the performance of both algorithms (without tuning) with other approaches. Section 6 presents the tuning of X-HYBRIDJOIN and evaluates its performance after tuning. Finally, Sect. 7 concludes the paper.

A description of an untuned version of X-HYBRIDJOIN has been published in BNCOD '11 [24]. The paper extends that work by providing a tuning approach for X-HYBRIDJOIN. Tuning is important for X-HYBRIDJOIN, since this algorithm is designed to be executed within limited resources and we want to make sure that these limited resources are used efficiently. Other similar join operators [26,27] employ a tuning approach as well, so it is important to understand tuning of X-HYBRIDJOIN.

2 Related work

In this section, we present an overview of the previous work that has been done in this area, focusing on that which is closely related to our problem domain.

The Non-blocking Symmetric Hash Join (SHJ) algorithm [32,33] extends the proprietary hash join algorithm in a pipeline fashion. In the symmetric hash join, there is a separate hash table for both input relations. When the tuple of one input arrives, it probes into the hash table of the other input, generates a result and then stores it in its own hash table. SHJ can produce the result before reading either input relation entirely; however, it stores both the relations in memory.

The Double-Pipelined Hash Join (DPHJ) [13], an extension of SHJ, is a two-stage join algorithm. The first stage is executed in memory and is similar to SHJ, while the second stage is disk-based and the additional tuples that are not joined in memory are joined on disk. But duplicate tuples are possible during the cleanup phase, which starts after both inputs have been completed, and some flushing policy is also required to flush the tuples on disk.

The XJoin algorithm [29,30] is another extension of SHJ. This is basically a three-stage join algorithm. The first stage begins when the tuple from either input source becomes available. In the case where both inputs are blocked, the second stage starts and finally the third stage executes when the cleanup join starts after receiving all the inputs. To eliminate the duplication of tuples, XJoin uses a time stamp approach, but still there is a chance of duplication and also a flushing policy is required in order to transfer the extra tuples to the disk.

Hash-Merge Join (HMJ) [21] is also from the series of symmetric joins. This is based on push technology and consists of two phases, hashing and merging.

MJoin [31], a generalized form of XJoin, extends the symmetric binary join operators to handle multiple inputs. MJoin uses a separate hash table for each input. On the arrival of a tuple from an input, it is stored in the corresponding hash table and is probed in the rest of the hash tables. It is not necessary to probe all the hash tables for each arrival, as the sequence of probing stops when a probed tuple does not match in a hash table. The methodology for choosing the correct sequence of probing is determined by performing the most selective probes first. The algorithm uses a coordinated flushing technique that involves flushing the same partition on the disk for all inputs. The three stages are similar to the stages in XJoin and MJoin. To identify the duplicate tuples, MJoin uses two timestamps for each tuple, the arrival time and the departure time from memory.

Early Hash Join (EHJ) [19] is a further extension of XJoin. EHJ introduced a new biased flushing policy that flushes the partitions of the largest input first. EHJ also simplified the strategies for determining the duplicate tuples based on cardinality; therefore, no timestamps are required for the arrival and departure of input tuples. However, EHJ is based on pull technology, and therefore, a reading policy is required for inputs.

Progressive Merge Join (PMJ) algorithm [6] is an enhanced version of the traditional sort-merge join. To enhance the performance of PMJ, a novel flushing algorithm was proposed by Tao et al [28]. But this algorithm again stores the extra stream tuples on disk rather than in memory, and therefore, it is expensive to refresh them in an online fashion in order to process the continuous data stream.

A novel algorithm Mesh Join (MESHJOIN) [26,27] has been designed especially for joining a continuous stream with a disk-based relation, like the scenario in active data warehouses. Although it is an adaptive approach, there are some research issues such as suboptimal distribution of memory among the join components and an inefficient strategy for accessing the disk-based relation.

A revised version of MESHJOIN called R-MESHJOIN (reduced Mesh Join) [23] has been presented that addresses the issue of optimal distribution of memory among the join components. In this algorithm a new strategy for memory distribution among the join components is introduced by implementing real constraints. However, the issue of an inefficient strategy for accessing the disk-based relation still exists in R-MESHJOIN.

One approach for improving MESHJOIN has been a partition-based join algorithm [5] which can also deal with stream intermittence. It uses a two-level hash table in order to attempt to join stream tuples as soon as they arrive and uses a partition-based waiting area for other stream tuples. For the algorithm in [5], however, the time that a tuple waits for execution is not bounded. We are, however, interested in a join approach where there is a time guarantee for when a stream tuple will be joined.

Another recent approach Semi-Streaming Index Join (SSIJ) [3] has been developed to process stream data with disk-based data. In general, the algorithm is divided into three phases, namely the pending phase, the online phase and the join phase. In the pending phase the stream tuples wait in a buffer until the size of the buffer is less than the predefined threshold limit or the stream ends. Once the size of the buffer crosses the threshold limit or the stream ends, the algorithm starts its online phase. In the online phase stream tuples from the input buffer are looked up in cache-based disk blocks. If the required disk tuple exists in cache, the join is executed and the algorithm produces an output. In the case where the required disk tuple is not available in cache, the algorithm flushes the stream tuple into a stream buffer where it waits for the join phase. The algorithm implements another threshold on the stream buffer, and during the join phase, the disk invoke is performed if the size of the stream buffer is greater than the threshold value. In the join phase the algorithm joins the tuples from the stream buffer with the tuples in the invoked disk block. The algorithm

implements a utility counter for each disk block that determines which disk blocks need to be retained in memory.

Although SSIJ is a feasible approach for processing stream data, the algorithm does not include a mathematical cost model. Consequently, it makes the criteria for calculating threshold parameters unclear. In addition, the architecture that SSIJ uses is quite complex compared to that we propose for X-HYBRIDJOIN.

3 Index-based hash join architecture: HYBRIDJOIN

In this section we introduce the HYBRIDJOIN algorithm, which implements our first modification of MESHJOIN in order to make use of a non-clustered index. We introduce the join architecture for HYBRIDJOIN. This will be used, with a single modification, for the X-HYBRIDJOIN algorithm as well.

HYBRIDJOIN joins a disk-based relation R with a stream S. We assume a non-clustered index on R for the join attribute, and we assume that the join attribute is unique within the master data. This is a very natural set of assumptions and matches with common application domains, for example, in key exchange applications. By only requiring a non-clustered index, we keep our assumptions as minimal as possible.

The memory architecture used in HYBRIDJOIN and in X-HYBRIDJOIN is shown in Fig. 2. The main memory components are a disk buffer, a hash table, a queue and a stream buffer while the disk-based relation R and stream S are the inputs. In our algorithm, we assume that R has an index with the join attribute as the key. The stream is used as the build input. This means that the algorithm keeps stream tuples in a hash table which occupies the largest share of the memory, and the hash table is filled with the next pending stream tuples to its full capacity. Additionally, we keep identifiers of the stream tuples in a queue, which allows random deletion; the simplest implementation is a doubly linked list.

HYBRIDJOIN is an iterative algorithm, and in each iteration it uses a partition of the disk-based relation R as a probe input. For that purpose, the partition is loaded into the disk buffer. In HYBRIDJOIN, the disk buffer contains only this partition; later in X-HYBRIDJOIN the partition will only occupy one part of the disk buffer. After that, the algorithm performs the typical operation of a hash join, i.e., it loops over all the tuples of the disk buffer and looks them up in the hash table. In the case of a match, the algorithm generates an output.

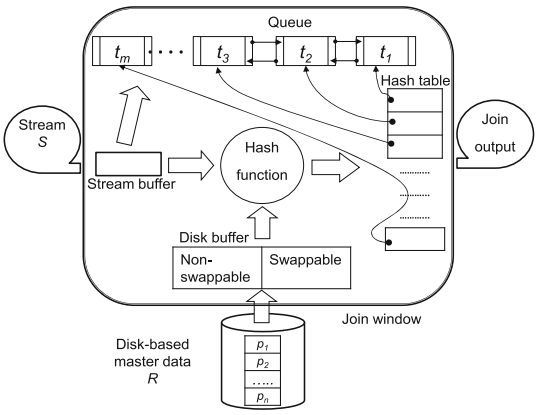


Fig. 2 Architecture of HYBRIDJOIN and X-HYBRIDJOIN. The only difference between the two algorithms is that in X-HYBRIDJOIN the disk buffer is split and its two parts are treated differently, as explained in the text

Also, in each iteration, HYBRIDJOIN evicts stream tuples that have been matched. This is justified through the assumption that the join attribute is unique in R. Evicting a tuple means it is deleted from the hash table and the queue. The algorithm also keeps a counter w of the evicted tuples. After processing the whole disk buffer, the algorithm reads w new tuples from the stream buffer and loads them in the hash table while also entering their identifiers in the queue.

For choosing the next partition of R, HYBRIDJOIN looks at the join attribute of the oldest stream tuple in the queue. Using the index, it loads the partition of R with that join attribute value into the disk buffer. It is this last step which makes HYBRIDJOIN adaptive, because in HYBRIDJOIN, every loaded partition matches at least one stream tuple. As a simple example, consider R has a section that is not referred to in the stream, for example, an obsolete group of products. In MESHJOIN, this section would still be loaded, while in HYBRIDJOIN it would not be loaded, because no stream tuple will trigger the loading of that section. HYBRIDJOIN works for any data distribution, as MESHJOIN does. However, in practice, certain distributions are common. Current research has shown that sales data typically follow a power law, or Zipfian distribution [2]. The power law is characterized by its exponent. For an exponent <1, the distribution is said to have a long tail, and for an exponent >1, the distribution has a short tail. For exponent 1 we get the distribution of Zipf's law [20], which gave rise to the general term Zipfian distribution. In sales, the 80/20 rule is used to model the scenario where the frequency of selling a small number of products is significantly higher compared to the rest of the products, often simplified in the 80/20 rule [15]. The 80/20 rule of thumb has commonly been observed in commercial applications [11,15]. The 80/20 rule corresponds to an exponent slightly smaller than 1 [15].

4 X-HYBRIDJOIN

In this section, we describe our second algorithm X-HYBRIDJOIN, which is an extension of HYBRIDJOIN.

4.1 Difference between X-HYBRIDJOIN and HYBRIDJOIN

As we will see later, the service rate (number of tuples processed per second) of HYBRIDJOIN increases as the exponent of the distribution goes above 1, i.e., as the distribution gets closer to a short-tailed distribution. However, if a distribution is fairly short-tailed, then many matches are with the most frequent tuples. So the question arises is how much can be gained in terms of performance by buffering the most frequent tuples permanently, and this gives rise to X-HYBRIDJOIN.

The difference between the two algorithms is that in X-HYBRIDJOIN we add a new part to the disk buffer. This part stores the most popular pages of disk-based relation R in memory permanently, and we call this the non-swappable part of the disk buffer. In X-HYBRIDJOIN, as a rule of thumb, the size of the new non-swappable part is initially the same as the swappable disk buffer in HYBRIDJOIN. However, in Sect. 6 the relative size of both parts will be optimized. Since we assume that the data are sorted according to access frequency, we know that the first pages in the relation R are the most popular pages. The other part of the disk buffer is swappable and is used to load partitions from the remainder of relation R into memory in the same way as in the HYBRIDJOIN algorithm.

Algorithm 1 Pseudo-code for X-HYBRIDJOIN

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

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

4.2 Algorithm

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

Table 1 Notations used in cost estimation of X-HYBRIDJOIN

Parameter name	Symbol
Total allocated memory (bytes)	M
Service rate (or performance) (processed tuples/sec)	μ
Input size (=number of matching tuples in previous iteration)	w
Stream tuple size (bytes)	v_S
Size of each swappable and non-swappable part (bytes) (=size of 1 disk partition)	v_P
Size of disk tuple (bytes)	v_R
Size of each swappable and non-swappable part (tuples)	$d_T = \frac{v_P}{v_R}$
Memory weight for the hash table	α
Memory weight for the queue	$1 - \alpha$
Cost to read one disk partition into the disk buffer (nanosecs)	$c_{I/O}(v_P)$
Cost to look up one tuple in the hash table (nanosecs)	c_H
Cost to generate the output for one tuple (nanosecs)	c_O
Cost to remove one tuple from the hash table and the queue (nanosecs)	c_E
Cost to read one stream tuple into the stream buffer (nanosecs)	c_S
Cost to append one tuple into the hash table and the queue (nanosecs)	c_A
Total cost for one loop iteration of X-HYBRIDJOIN (secs)	c_{loop}

hash table (lines 8–14). Before starting the second inner loop, the algorithm reads the oldest value of the join attribute from the queue and loads a disk partition p_i (where $2 \leq i \leq n$) into the swappable part of the disk buffer, using that join attribute value as an index (lines 15 and 16). As the specified disk partition is loaded into the swappable part of the disk buffer, the algorithm starts the second inner loop and repeats all the steps described in the first inner loop (lines 17–23).

Note: If we switch off the first inner loop (lines 8–14), the algorithm works as HYBRIDJOIN.

4.3 Cost model

In this section we derive the general formulae for calculating the cost for our proposed X-HYBRIDJOIN. We derive equations for memory and processing time of X-HYBRIDJOIN. Equation 1 describes the total memory used to implement the algorithm except for the stream buffer, whereas Eq. 2 calculates the processing cost for w tuples. The symbols used to measure the costs are specified in Table 1.

4.3.1 Memory cost

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

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

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

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

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

Currently, we do not include the memory reserved by the stream buffer because of its small size (0.05 MB has been sufficient in all our experiments).

4.3.2 Processing cost

In this section, we calculate the processing cost for X-HYBRIDJOIN. We denote the cost for one loop iteration of the algorithm as c_{loop} and express it as the sum of the costs for the individual operations. To make it simple, we first calculate the processing cost for each component separately.

Cost to read swappable or non-swappable parts of the disk buffer $= c_{I/O}(v_P)$
Cost to look-up swappable and non-swappable parts of the disk buffer in the hash table $= d_T c_H + d_T c_H = 2d_T c_H$
Cost to generate the output for w matching tuples $= w \cdot c_O$
Cost to remove w tuples from the hash table and the queue $= w \cdot c_E$
Cost to read w tuples from stream $S = w \cdot c_S$
Cost to append w tuples in the hash table and the queue $= w \cdot c_A$

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

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

For all c_{loop} seconds, the algorithm processes w tuples of stream S; therefore, the service rate (or performance) μ can be calculated by dividing w by the cost for one loop iteration.

$$\mu = \frac{w}{c_{\text{loop}}} \tag{3}$$

5 Experiments

In this section we describe the setup for the experiments and analyze the performance of our algorithm with other related algorithms. As mentioned before, we use synthetic datasets with a known skew.

5.1 Experimental setup

5.1.1 Hardware specifications

We carried out our experiments on a Pentium-IV 2 × 2.13 GHz machine with 3G main memory and 160G disk memory under WindowsXP. We implemented the algorithm in Java using the Eclipse IDE 3.3.1.1. To measure the memory and processing time, we used built-in plugins provided by Apache and Java API, respectively.

Table 2 Data specification

Parameter	value
Disk-based data	
Size of disk-based relation R	0.5–8 million tuples
Size of each tuple	120 bytes
Stream data	
Size of each tuple	20 bytes
Size of each node in queue	12 bytes
Benchmark	
Based on	Zipf's law
Characteristics	Bursty and self-similar

5.1.2 Data specifications

The synthetic workload that we used to test the algorithms was generated using Zipf's Law with exponent 1. The generated stream has two additional characteristics known as burstyness and self-similarity. The detailed specifications of the dataset that we used for analysis are shown in Table 2. The relation R is stored on disk using MySQL 5.0 databases. To measure the cost for each I/O operation accurately, we set the fetch size for the ResultSet equal to the size of one partition on disk. X-HYBRIDJOIN needs to store multiple values in the hash table against one key value. However, the hash table provided by the standard Java API does not support this feature; therefore, we have used the Multi-Hash-Map from Apache as the hash table in our experiments.

5.1.3 Measurement strategy

We define the performance of the algorithms as service rate, with a higher service rate being better. The service rate has been measured by calculating the number of tuples processed in a unit second. In each experiment, the algorithm is executed for 1 h. We started our measurements after 20 min and keep measuring for 20 min. For added accuracy, we took three readings for each specification and then calculated the average. Where required, we also calculated the confidence interval by considering 95 % accuracy. The calculation of confidence interval is based on 4,000 measurements for one setting. Moreover, during the execution of the algorithm, no other application was running in parallel.

5.2 Experimental results

In our experimental study, we analyzed the results from two different perspectives. Firstly, we compare the performance of both our algorithms, HYBRIDJOIN and X-HYBRIDJOIN, with the other related algorithms. Secondly, in X-HYBRIDJOIN we examine the role of the non-swappable part of the disk buffer in stream processing. The three possible parameters that can vary and directly affect the performance of the algorithms under test are the total available memory for the algorithm, the size of the disk-based relation R and the variation of skew in stream data. In our experiments, we tested the algorithms for different values of these parameters and compared their performance.

5.2.1 Size of available memory varies

In our first experiment, we analyzed the performance of X-HYBRIDJOIN using different memory budgets while the size of R is fixed (2 million tuples). Figure 3a presents the results of our experiment. The figure indicates that, for all memory budgets, the performance of X-HYBRIDJOIN is significantly better than that of all the other algorithms. The reason behind this improvement is our intuition about X-HYBRIDJOIN. In our calculations, introducing the non-swappable part in X-HYBRIDJOIN can save about 33 % of the disk I/O cost. Although keeping the non-swappable part in memory increases the look-up cost and reduces the memory for the hash table, both these factors are very small compared to the disk I/O cost.

From the experiments we can see that HYBRIDJOIN performs consistently slightly better than MESHJOIN and R-MESHJOIN. However, the improvement is rather modest. Our experiments show that the main performance gain of X-HYBRIDJOIN is due to the second improvement, the introduction of a non-swappable part in the disk buffer.

5.2.2 Size of R varies

In the experiment shown in Fig. 3b, we assumed the total allocated memory for the join was fixed while the size of the disk-based relation R was grown exponentially. Figure 3b shows that for all sizes of R performance of X-HYBRIDJOIN is substantially better than all the other approaches. Another key observation from the figure is that when R is 0.5 million, the performance of HYBRIDJOIN is almost 70 % of X-HYBRIDJOIN and when R is equal to 8 million, this percentage decreases to 50 %. This means that the performance of the other algorithms decreases more sharply compared to X-HYBRIDJOIN when R increases.

5.2.3 Skew varies

Finally, we evaluate the performance of X-HYBRIDJOIN while varying the skew in the 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. In this experiment we assume the sizes of both available memory and R are fixed, 50 MB and 2 million tuples, respectively. Figure 3c presents the results of our experiment. It is clear from the figure that under all values of e except 0, X-HYBRIDJOIN performs considerably better than all other approaches. Also, this improvement becomes more pronounced for increasing exponent values. We did not present data for exponents larger than 1, which would imply short tails. It is clear that for such short tails the trend continues, but our focus was on understanding the limitations of our own approach. For completely uniform data, X-HYBRIDJOIN performs worse than MESHJOIN by a constant factor. The plausible reason for this performance is that in the case of a fully uniform stream data the non-swappable part of the disk buffer does not contribute much in performance as compared to the processing cost that originates due to this component.

5.2.4 Role of the non-swappable part in stream processing

To get a better understanding of the role of the non-swappable part of the disk buffer, we performed an experiment where we counted the stream tuples which are processed using only the non-swappable part of the disk buffer. The results of this experiment are shown in Fig. 3d. As before, we set the size of the non-swappable part to be equal to the size of the

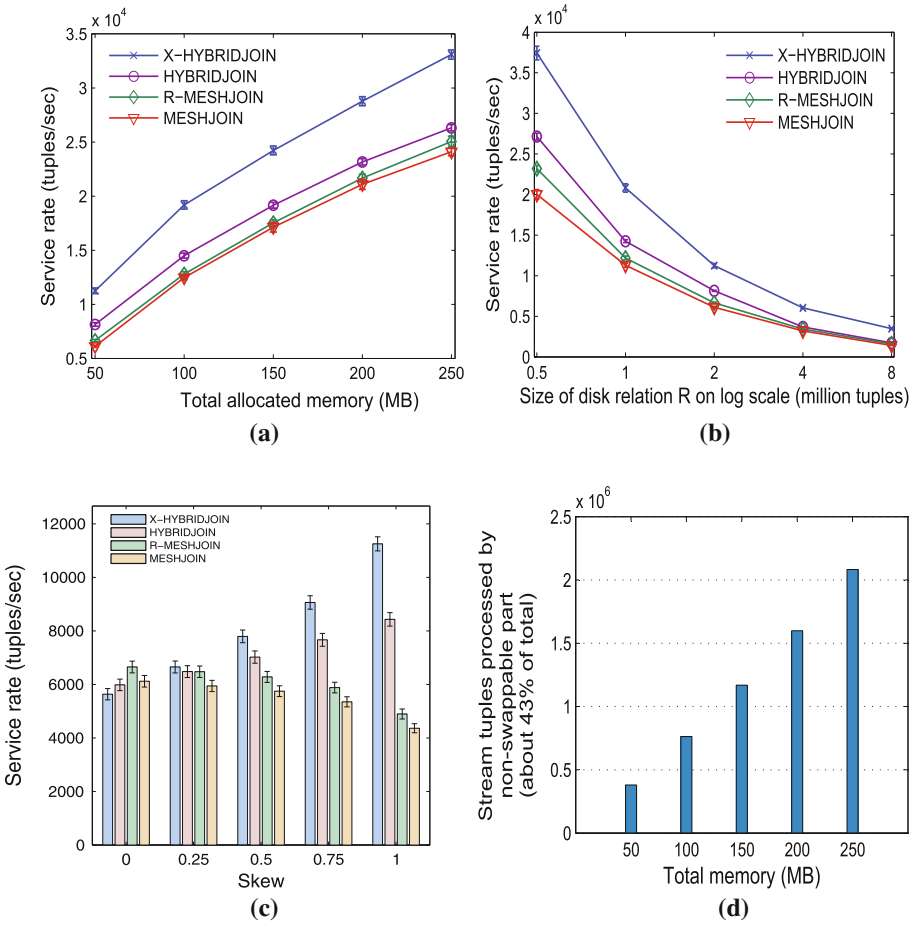


Fig. 3 Performance evaluation. **a** Memory size varies. **b** Size of R varies. **c** Skew varies. **d** Role of non-swappable part in performance

swappable part. It is clear from the figure that in 4,000 iterations when the memory budget is 50 MB and the size of R is 2 million tuples, about 0.4 million stream tuples are processed through the non-swappable part of the disk buffer and this number increases if we increase the total allocated memory. For 250 MB memory with the same size of R (2 million tuples), this amount reaches more than 2 million. In the other algorithms, since this non-swappable part is loaded from the disk each time, the I/O cost increases significantly.

6 Tuning

The tuning process, which is based on the cost model, is used to optimize the performance of the algorithm under resource constraints. Normally, if these kinds of algorithms are seen in isolation, having more memory available would be better for each component. However, assuming a fixed memory allocation provides a trade-off in the distribution of memory. Assigning more memory to one component means that less memory is available for other

components. Therefore, it is necessary to find the optimal distribution of memory among all components in order to attain maximum performance. A very important component here is the disk buffer because reading data from disk to memory is very expensive.

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

6.1 Revised cost model

In order to tune X-HYBRIDJOIN, it is first necessary to revise the cost model. The reason for this revision is that previously we assumed that both parts of the disk buffer were of equal size but in fact they can vary. Therefore, the formulas derived before do not apply for the tuning of the algorithm. Using a revised cost model, Eq. 4 describes the total memory used to implement the algorithm, while Eq. 5 calculates the processing cost for w tuples.

6.1.1 Memory cost

Since the optimal values for the sizes of both the swappable part and non-swappable part can be different, k number of pages is assumed for the swappable part and l number of pages for the non-swappable part. The memory for each component can be calculated as given below:

Memory for the swappable part of the disk buffer (bytes)$= kv_P$
Memory for the non-swappable part of the disk buffer (bytes)$= lv_P$
Memory for the hash table (bytes)$=\alpha[M-(k+l)v_P]$
Memory for the queue (bytes)$= (1-\alpha)[M-(k+l)v_P]$

The total memory used by the algorithm can be determined by aggregating the above.

$$M = (k+l)v_P + \alpha[M-(k+l)v_P] + (1-\alpha)[M-(k+l)v_P] \tag{4}$$

6.1.2 Processing cost

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

Cost to read the non-swappable part of the disk buffer (nanoseconds)$= c_{I/O}(l.v_P)$
Cost to read the swappable part of the disk buffer (nanoseconds)$= c_{I/O}(k.v_P)$
Cost to look up the non-swappable part of the disk buffer in the hash table (nanoseconds) $=d_N c_H$ where $d_N = l\frac{v_P}{v_R}$ is the size of the non-swappable part of the disk buffer in terms of tuples.
Cost to look up the swappable part of the disk buffer in the hash table (nanoseconds)$=d_S c_H$ where $d_S = k\frac{v_P}{v_R}$ is the size of the swappable part of the disk buffer in terms of tuples.
Cost to generate the output for w matching tuples (nanoseconds)$= w \cdot c_O$
Cost to delete w tuples from the hash table and the queue (nanoseconds)$= w \cdot c_E$
Cost to read w tuples from stream S into the stream buffer (nanoseconds)$= w \cdot c_S$

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

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

$$c_{\text{loop}}(\text{secs}) = 10^{-9}[c_{I/O}(k.v_P) + (d_N + d_S)c_H + w(c_O + c_E + c_S + c_A)] \quad (5)$$

If the algorithm processes w tuples in c_{loop} seconds, then the service rate μ can be calculated using Eq. 6 which is similar to Eq. 3.

$$\mu = \frac{w}{c_{\text{loop}}} \quad (6)$$

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

6.2 Tuning using empirical approach

This section focuses on obtaining samples for the approximate tuning of the key components. The performance is a function of two variables, the size of the swappable part of the disk buffer, d_S, and the size of the non-swappable part of the disk buffer, d_N. The performance of the algorithm has been tested for a grid of values for both components, i.e., for each setting of d_S the performance was measured against a series of values for d_N. The performance measurements for the grid of d_S and d_N are shown in Fig. 4. The figure shows that the performance increases rapidly as the size for the non-swappable part increases. After reaching a particular value for the size of the non-swappable part, the performance starts decreasing. The plausible reason behind this behavior is that in the beginning when the size for the non-swappable part increases, the probability of matching stream tuples with disk tuples also increases and that improves the performance. But when the size for the non-swappable part is increased further, it does not make a significant difference in stream-matching probability due to the factor of skew in distribution. The higher look-up cost associated with the increased non-swappable part and the fact that less memory is available for the hash table means that the performance gradually decreases.

A similar behavior has been observed when the performance has been tested for the swappable part. Initially, the performance increases, since the costly disk access is amortized

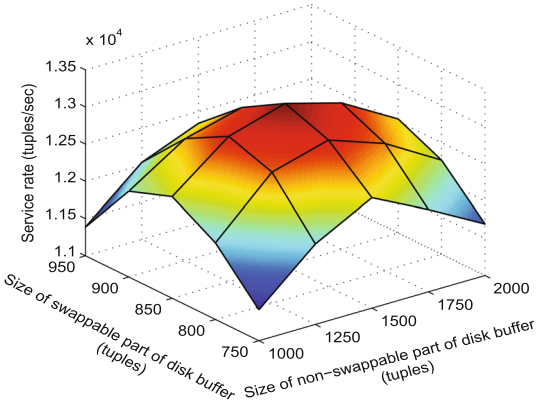


Fig. 4 Tuning of X-HYBRIDJOIN using empirical approach

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

From the measurements shown in Fig. 4, it is possible to approximate the optimal settings for both the swappable and the non-swappable parts by considering the intersection of the values of both components at which the algorithm individually performs at a maximum.

6.3 Tuning based on cost model

A mathematical model for the tuning is also derived based on the cost model presented in Sect. 6.1. From Eq. 6 it is clear that the service rate depends on the size of w and the cost c_{loop}. To determine the optimal settings, it is first necessary to calculate the size of w. The value of w in X-HYBRIDJOIN is the total number of stream tuples which match with both the swappable and non-swappable parts in each iteration. Before deriving a mathematical formula for w, the main components that can affect w are discussed. The main components on which the value of w depends are listed below.

1. Size of non-swappable part, d_N
2. Size of swappable part, d_S
3. Size of disk-based relation, R_t
4. Size of hash table, h_S

Typically, the stream of updates can be approximated using Zipf's law with a certain exponent value. Therefore, a significant part of the stream is joined with the non-swappable part of the disk buffer. Hence, if the size of the non-swappable part (i.e., d_N) is increased, more stream tuples will match as a result. But the probability of matching does not increase at the same rate as increasing d_N because, according to the Zipfian distribution, the matching probability for the second tuple in R is half of that for the first tuple and similarly the matching probability for the third tuple is one-third of that for the first tuple and so on [2,15]. Due to this property, the size of R (denoted by R_t) also affects the matching probability. The swappable part of the disk buffer deals with the rest of the disk-based relation denoted by R' (where $R' = R_t - d_N$), which is less frequent in the stream than that part which exists permanently in memory. The algorithm reads R' in partitions, where the size of each partition is equal to the size of the swappable part of the disk buffer d_S. In each iteration the algorithm reads one partition of R' using an index on join attribute and loads it into memory through a swappable part of the disk buffer. In the next iteration the current partition in memory is replaced by a new partition, and so on. As mentioned earlier, using the Zipfian distribution, the matching probability for each tuple is less than the previous one. Therefore, the total number of matches against each partition is not the same. This is explained further in Fig. 5, where n total partitions are considered in R'. From the figure it can be seen the matching probability for each disk partition decreases continuously as we move toward the end position in R. The size of the hash table is another component that affects w. The reason is that if there are more stream tuples in memory, the number of matches will be greater and vice versa.

Before deriving the formula to calculate w, it is first necessary to understand the working strategy of X-HYBRIDJOIN. Consider for a moment that the queue contains stream tuples instead of just join attribute values. It has already been stated in Sect. 4.2 that X-HYBRIDJOIN uses two independent inner loops under one outer loop. After the end of the first inner loop, which means after finishing the processing of the non-swappable part, the queue only contains

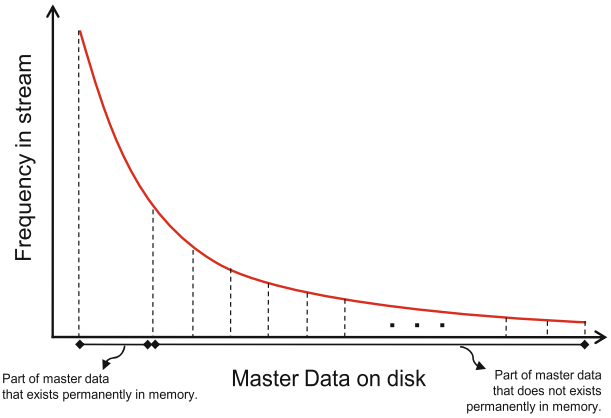


Fig. 5 A sketch of matching probability of R in stream

those stream tuples which are related to only the swappable part of R, denoted by R'. For the next outer iteration of the algorithm, these stream tuples in the queue are considered to be an old part of the queue. In that next outer iteration the algorithm loads some new stream tuples into the queue and these new stream tuples are considered to be the new part of the queue. The reason for dividing the queue into two parts is that the matching probability for both parts of the queue is different. The matching probability for the old part of the queue is denoted by p_{old}, and it is only based on the size of the swappable part of R, i.e., R'. On the other hand, the matching probability for the new part of the queue, known as p_{new}, depends on both the non-swappable and the swappable parts of R. Therefore, to calculate w we first need to calculate both these probabilities.

Therefore, if the stream of updates S obeys Zipf's law where the exponent value is equal to 1, then the matching probability for any swappable partition m with the old part of the queue can be determined mathematically as shown below.

$$p_m = \frac{\sum_{x=d_N+(m-1)d_S+1}^{d_N+md_S} \frac{1}{x}}{\sum_{x=d_N+1}^{R_t} \frac{1}{x}}$$

Each summation in the above equation generates a harmonic series, which can be summed up using the formula $\sum_{x=1}^{m} \frac{1}{x} = \ln m + \gamma + \epsilon_m$ where γ is Euler's constant [1] whose value is approximately equal to 0.5772156649 and ϵ_m is another constant which is $\approx \frac{1}{2m}$. The value of ϵ_m approaches 0 as m goes to ∞ [1]. In our case the value of $\frac{1}{2m}$ is small and therefore, it is ignored. If there are n partitions in R', then the average probability of an arbitrary partition of R' matching the old part of the queue can be determined using Eq. 7.

$$\overline{p}_{\text{old}} = \frac{\sum_{m=1}^{n} p_m}{n} = \frac{1}{n} \tag{7}$$

Now the probability of matching is determined for the new part of the queue. Since the new input stream tuple can match either the non-swappable or the swappable part of R, the average matching probability of the new part of the queue with both parts of the disk buffer can be calculated using Eq. 8.

$$\overline{p}_{\text{new}} = p_N + \frac{1}{n} p_S \tag{8}$$

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

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

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

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

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

If w is the average number of matches per outer iteration with both the swappable and non-swappable parts in the disk buffer, then w can be calculated by applying the binomial probability distribution on Eqs. (7) and (8) as given below.

$$w = (h_S - w)p_{\text{old}}(1 - p_{\text{old}}) + wp_{\text{new}}(1 - p_{\text{new}})$$

After simplification, the final formula to calculate w is described in Eq. 9.

$$w = \frac{h_S p_{\text{old}}(1 - p_{\text{old}})}{1 + p_{\text{old}}(1 - p_{\text{old}}) - p_{\text{new}}(1 - p_{\text{new}})} \tag{9}$$

6.4 Comparisons of both tuning approaches

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

6.4.1 Swappable part

In this experiment the tuning results have been compared for the swappable part of the disk buffer using both the measurement and the cost model approaches. The tuning results for each approach (with a 95 % confidence interval in the case of the empirical approach) are shown in Fig. 6a. It is evident that at every position the results in both cases are similar, with only 0.5 % deviation.

6.4.2 Non-swappable part

Similarly, the tuning results of both approaches have been compared for the non-swappable part of the disk buffer. The results are shown in Fig. 6b. Again, it is clear from the figure that the results in both cases are nearly equal, with a deviation of only 0.6 %.

6.5 Performance evaluation after tuning

6.5.1 Performance comparisons for different sizes of R

In these experiments the performance of X-HYBRIDJOIN has been compared with and without tuning while varying the size of R. It is assumed that the size of R varies exponentially

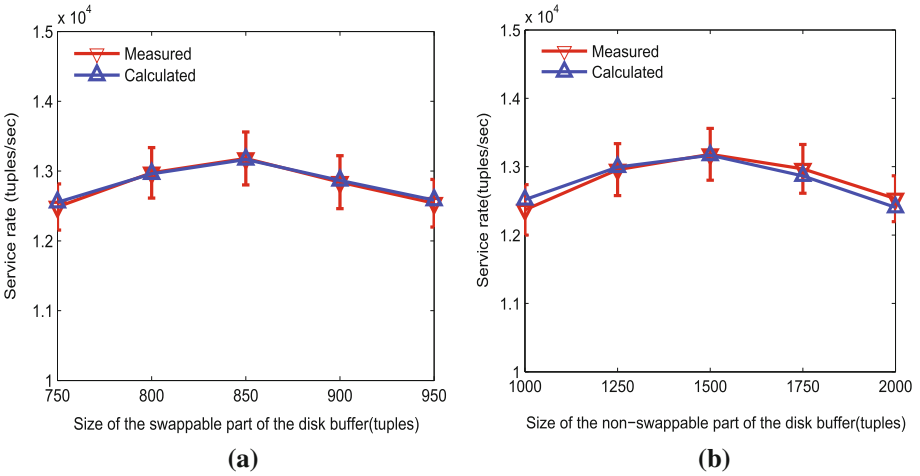


Fig. 6 Comparisons of tuning results. **a** Tuning comparison for swappable part: based on measurements versus based on cost model. **b** Tuning comparison for non-swappable part: based on measurements versus based on cost model

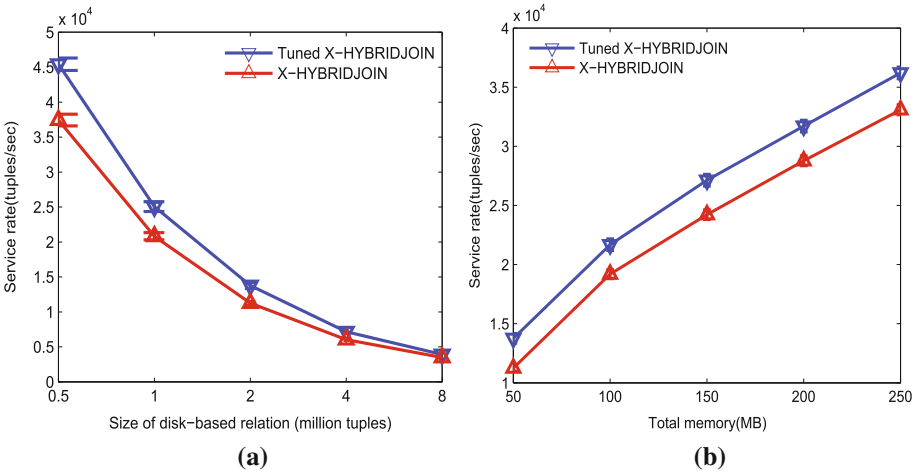


Fig. 7 Performance comparisons: Tuned X-HYBRIDJOIN versus X-HYBRIDJOIN without tuning. **a** Size of disk-based relation varies. **b** Total allocated memory varies

while the total memory budget remains fixed (50 MB) for all values of R. For each value of R the X-HYBRIDJOIN algorithm has been run both with and without optimal settings, and the performance has been measured in both cases. The performance results that have been obtained using both experiments are shown in Fig. 7a. It is clear that for all settings of R, X-HYBRIDJOIN performed significantly better with tuning than without.

6.5.2 Performance comparisons for different memory budgets

In these experiments the performance in both cases has been compared using different memory budgets while the size of R is fixed (2 million tuples). Figure 7b depicts the comparisons

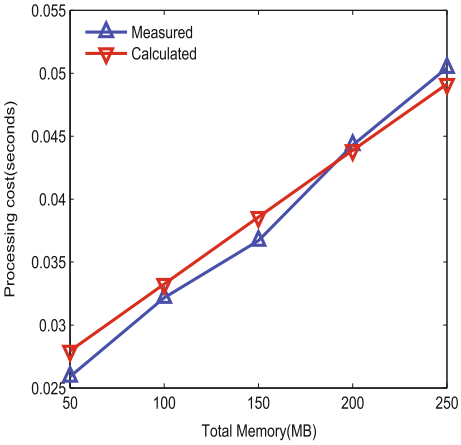


Fig. 8 Cost validation for X-HYBRIDJOIN

of both cases. It can be observed that for all memory budgets the tuned X-HYBRIDJOIN again performed significantly better than simple X-HYBRIDJOIN.

6.6 Cost validation

The cost model for X-HYBRIDJOIN has been validated by comparing the calculated cost with the measured cost. Figure 8 presents the comparisons of both costs. The figure shows that the calculated cost closely resembles the measured cost, which proves the correctness of the cost model.

7 Conclusions and future directions

To keep data warehouses up-to-date in order to support businesses fully, a near-real-time data feed is required. In the transformation phase, a join operator is required to perform a continuous join between the fast stream and the disk-based relation within limited memory resources. In this paper, we explored the potential improvement for stream-based joins if characteristics of the data such as skew are taken into account. MESHJOIN performs worse with skewed distributions, which is a problem since these distributions are common in real-world applications. We presented a robust algorithm called X-HYBRIDJOIN (Extended Hybrid Join) with two major variations over MESHJOIN. The first variation is the use of an index on disk-based master data. The second variation is that X-HYBRIDJOIN caches the most frequent tuples of master data. As a result, it reduces the disk access and improves the performance substantially. We also calculated the cost model for X-HYBRIDJOIN and tuned the algorithm based on that cost model. To validate our arguments, we implemented the prototypes for both modifications and carried out experiments comparing the different algorithms. We provided open source implementations of our algorithm.

In the future we plan to propose a generalized algorithm for the processing of stream data with disk-based data. The main aim of this algorithm will be to remove the need for sorting the disk-based master data.

References

1. Abramowitz M, Stegun IA (1964) Handbook of mathematical functions with formulas, graphs, and mathematical tables. Dover, New York
2. Anderson C (2006) The long tail: why the future of business is selling less of more. Hyperion
3. Bornea MA, Deligiannakis A, Kotidis Y, Vassalos V (2011) Semi-streamed index join for near-real time execution of ETL transformations. In: ICDE '09: proceedings of the 27th international conference on data engineering (ICDE). IEEE Computer Society, Washington, DC, USA, pp 159–170
4. Bruckner RM, List B, Schiefer J (2002) Striving towards near real-time data integration for data warehouses. In: DaWaK 2000: proceedings of the 4th international conference on data warehousing and knowledge discovery. Springer, London, UK, pp 317–326
5. Chakraborty A, Singh A (2009) 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 and distributed processing. IEEE Computer Society, Washington, DC, USA, pp 1–11
6. Dittrich J, Seeger B, Taylor DS, Widmayer P (2002) Progressive merge join: a generic and non-blocking sort-based join algorithm. In: VLDB '02: proceedings of the 28th international conference on very large data bases. Hong Kong, China, pp 299–310
7. Francisco A (2003) Real-time data warehousing with temporal requirements. In: CAiSE workshops
8. Golab L, Johnson T, Seidel JS, Shkapenyuk V (2009) Stream warehousing with datadepot. In: SIGMOD '09: proceedings of the 35th SIGMOD international conference on management of data. ACM, New York, NY, USA, pp 847–854
9. Gupta A, Mumick IS (1995) Maintenance of materialized views: problems, techniques, and applications. IEEE Data Eng Bull 18(2):3–18
10. Han X, Li J, Yang D (2012) PI-join: efficiently processing join queries on massive data. Knowl Inf Syst 32(3):527–557
11. Heising WP (1963) Note on random addressing techniques.: IBM Syst J **2**(2), 112–116
12. Hohpe G, Woolf B (2003) Enterprise integration patterns: designing, building, and deploying messaging solutions. Addison-Wesley Longman Publishing, Boston
13. Ives ZG, Florescu D, Friedman M, Levy A, Weld DS (1999) An adaptive query execution system for data integration. In: SIGMOD Rec., vol 28, no 2. ACM, New York, NY, USA, pp 299–310
14. Karakasidis A, Vassiliadis P, Pitoura E (2005) ETL queues for active data warehousing. In: IQIS '05: proceedings of the 2nd international workshop on information quality in information systems. ACM, New York, NY, USA, pp 28–39
15. Knuth DE (2006) The art of computer programming, vol 3, 2nd edn. Sorting and searching. Addison Wesley Longman Publishing, Redwood City
16. Labio W, Garcia-Molina H (1996) Efficient snapshot differential algorithms for data warehousing. In: VLDB '96: proceedings of the 22th international conference on very large data bases. San Francisco, CA, USA, pp 63–74
17. Labio W, Yang J, Cui Y, Garcia-Molina H, Widom J (2000) Performance issues in incremental warehouse maintenance. In: VLDB '00: proceedings of the 26th international conference on very large data bases. San Francisco, CA, USA, pp 461–472
18. Labio WJ, Wiener JL, Garcia-Molina H, Gorelik V (2000) Efficient resumption of interrupted warehouse loads. In: SIGMOD Rec., vol 29, no 2. New York, NY, USA, pp 46–57
19. Lawrence R (2005) 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. VLDB endowment, Trondheim, Norway, pp 841–852
20. Levene M, Borges J, Loizou G (2001) Zipf's law for web surfers. Knowl Inf Syst 3(1):120–129
21. Mokbel MF, Lu M, Aref WG (2004) Hash-merge join: a non-blocking join algorithm for producing fast and early join results. In: ICDE '04: proceedings of the 20th international conference on data engineering. IEEE Computer Society, Washington, DC, USA, pp 251–263
22. Naeem MA, Dobbie G, Weber G (2008) An event-based near real-time data integration architecture. In: Enterprise distributed object computing conference workshops. IEEE, Munich, Germany, pp 401–404
23. Naeem MA, Dobbie G, Weber G (2010) R-MESHJOIN for near-real-time data warehousing. In: DOLAP'10: proceedings of the ACM 13th international workshop on data warehousing and OLAP. ACM, Toronto, Canada, pp 53–60
24. Naeem MA, Dobbie G, Weber G (2011) X-HYBRIDJOIN for near-real-time data warehousing. In: Proceedings of 28th British national conference on databases (BNCOD 28). Springer, Berlin/Heidelberg, pp 33–47

25. Nguyen A, Tjoa A (2003) Zero-latency data warehousing for hetrogeneous data sources and continuous data streams. In: iiWAS'2003: the fifth international conference on information integrationand web-based applications services, Austrian Computer Society (OCG), pp 55–64
26. Polyzotis N, Skiadopoulos S, Vassiliadis P, Simitsis A, Frantzell N (2008) Meshing streaming updates with persistent data in an active data warehouse. IEEE Trans Knowl Data Eng 20(7):976–991
27. Polyzotis N, Skiadopoulos S, Vassiliadis P, Simitsis A, Frantzell NE (2007) Supporting streaming updates in an active data warehouse. In: ICDE 2007. IEEE 23rd international conference on data engineering. Los Alamitos, CA, USA, pp 476–485
28. Tao Y, Yiu ML, Papadias D, Hadjieleftheriou M, Mamoulis N (2005) RPJ: producing fast join results on streams through rate-based optimization. In: SIGMOD '05: proceedings of the 2005 ACM SIGMOD international conference on management of data. New York, NY, USA. pp 371–382
29. Tolga U, Michael JF (2000) Xjoin: a reactively-scheduled pipelined join operator. IEEE Data Eng Bull 23(2):27–33
30. Urhan T, Franklin MJ (1999) XJoin: getting fast answers from slow and bursty networks. University of Maryland, College Park
31. Viglas SD, Naughton JF, Burger J (2003) Maximizing the output rate of multi-way join queries over streaming information sources. In: VLDB '2003: proceedings of the 29th international conference on very large data bases. VLDB Endowment, Berlin, Germany, pp 285–296
32. Wilschut AN, Apers PMG (1991) Dataflow query execution in a parallel main-memory environment. In: PDIS '91: proceedings of the first international conference on parallel and distributed information systems. IEEE Computer Society Press, Los Alamitos, CA, USA, pp 68–77
33. Wilschut AN, Apers PMG (1990) Pipelining in query execution. In: PARBASE-90: international conference on databases, parallel architectures and their applications. Miami, FL, USA, pp 562–562
34. Zhang X, Rundensteiner EA (2002) Integrating the maintenance and synchronization of data warehouses using a cooperative framework. Inf Syst 27(4):219–243
35. Zhuge Y, García-Molina H, Hammer J, Widom J (1995) View maintenance in a warehousing environment. In: SIGMOD '95: proceedings of the 1995 ACM SIGMOD international conference on management of data. ACM, New York, NY, USA, pp 316–327

Author Biographies

Dr M. Asif Naeem is presently a lecturer in School of Computing and Mathematical Sciences, Auckland University of Technology, Auckland, New Zealand. He received his Ph.D. degree in Computer Science from The University of Auckland, New Zealand. He has been awarded a best Ph.D. thesis award from the University of Auckland. Before that Asif has done his Master's degree with distinction in the area of Web Mining. He has about twelve-year research, industrial and teaching experience. He has published over 30 research papers in peer-reviewed journals, conferences and workshops including IEEE, ACM and VLDB. His research interests include databases, data warehousing, data stream processing and query optimization.

Dr Gillian Dobbie is a professor in the Department of Computer Science at the University of Auckland, New Zealand. She received a Ph.D. from the University of Melbourne, an M.Tech.(Hons) and B.Tech.(Hons) in Computer Science from Massey University. She has lectured at Massey University, the University of Melbourne and Victoria University of Wellington and held visiting research positions at Griffith University and the National University of Singapore. Her research interests include formal foundations for databases, object-oriented databases, semi-structured databases, logic and databases, data warehousing, data mining, access control, e-commerce and data modeling. She has published over 100 international refereed journal and conference papers.

Dr Gerald Weber is a senior lecturer in the Department of Computer Science at the University of Auckland. He joined the University of Auckland in 2003. Gerald holds a Ph.D. from the FU Berlin. He is information director of the Proceedings of the VLDB Endowment, and he has been program chair of several conferences. He is a coauthor of the book "Form-Oriented Analysis" and of over 40 peer-reviewed publications. His research interests include databases and data models, human–computer interaction and theory of computation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.