X-HYBRIDJOIN for Near-Real-Time Data Warehousing

M. Asif Naeem, Gillian Dobbie, and Gerald Weber

Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand
mnae006@aucklanduni.ac.nz
{gill,gerald}@cs.auckland.ac.nz

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 Mesh Join (MESHJOIN) has been proposed to amortize disk access. MESHJOIN makes no assumptions about the data distribution. In real world applications, however, skewed distributions can be found, e.g, certain products are sold more frequently than the remainder of the products. The question arises, how much does MESHJOIN loose 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 used by non-adaptive approaches such as MESHJOIN.

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

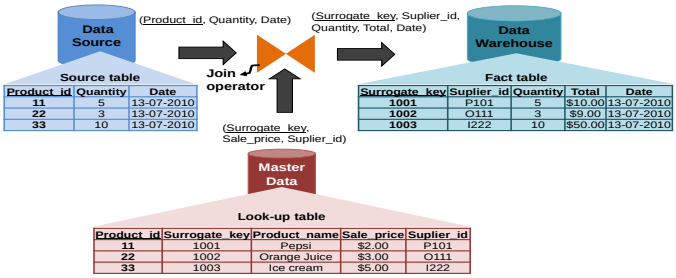
1 Introduction

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 [1] [2]. In the beginning, most data warehouses refreshed all content fully during each load cycle. However, due to an increasing demand for information freshness, it became infeasible to meet business needs. Therefore the data acquisition mechanism in warehouses was changed from full

refreshment to an incremental refresh strategy, in which new data is added to the warehouse without requiring a complete reload [3] [4]. Although this strategy is more efficient than the traditional one, it is still batch-oriented as a fraction of the data is propagated towards 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 [5] [6]; that is, sales 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. Common examples of such transformations are the enrichment of source data with master data attributes and the replacement of a *source data key* with a *warehouse key*. **Content enrichment** is a special form of data translation in which additional information is injected into the current message [7]. We consider an example of an inventory sales system, as shown in Figure 1. The *source table* contains attributes *product_id*, *quantity* and *date* with the *primary key* on attribute *product_id*. The *look-up table* that stores master data contains attributes *product_id*, *surrogate_key*, *product_name*, *sale_price* and *supplier_id* while having the *primary key* on attribute *product_id*. The attribute *surrogate_key* is the key for the data warehouse. This is used to identify the records uniquely and it is different in format from the source data key. Generally, the source updates are propagated with the attributes *product_id*, *quantity* and *date*. However, before loading these source updates into the data warehouse, they need to enrich with certain information from the *look-up table*. In our example, the enriched attributes are *supplier_id* and *total*. A join operator is required to perform these enrichment and key replacement tasks. 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 input update stream is high volume and has a bursty nature while 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 that high volume update stream with the slow disk access rate. 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 still there are a number of scenarios 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 is incoming. In the limited-memory approaches here, in contrast there is no such waste of resources.

A novel algorithm Mesh Join (MESHJOIN) [8] [9] 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 by the MESHJOIN authors have shown that the algorithm performs worse with skewed data. The MESHJOIN algorithm is a hash join, where the stream serves as the



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

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 such as suboptimal distribution of memory among the join components and an inefficient strategy for accessing the disk-based relation [12].

Although the MESHJOIN algorithm efficiently amortizes the fast input stream, the question remains how much potential for improvement remains untapped because the algorithm cannot 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 a select few products are bought with higher frequency [11]. 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 most 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.

X-HYBRIDJOIN has two major modifications compared with MESHJOIN. Firstly, the hash join component of X-HYBRIDJOIN 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. Since our purpose is primarily to gauge the potential of skewed distributions, we consider a very clean, artificial dataset that exactly exhibits a well-understood type of skew, a power law.

The remainder of the paper is structured as follows. A review of the related work is presented in Section 2. In section 3 we describe the intermediate algo-

rithm HYBRIDJOIN that already uses an index in the join process. In Section 4, we present the difference between HYBRIDJOIN and X-HYBRIDJOIN, and also derive the cost model for X-HYBRIDJOIN. The experimental study is discussed in Section 5 and finally Section 6 concludes the paper.

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.

A novel algorithm Mesh Join (MESHJOIN) [8] [9] has been designed especially for joining a continuous stream with a disk-based relation, such as 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) [12] has been presented by us. It 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 capturing 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 [10] 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 [10], however, the time that a tuple waits for execution is not bounded. We are, however, interested in a join approach where the time in which a stream tuple is joined is guaranteed.

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 algorithm X-HYBRIDJOIN 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 the application domain described above, 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 Figure 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.

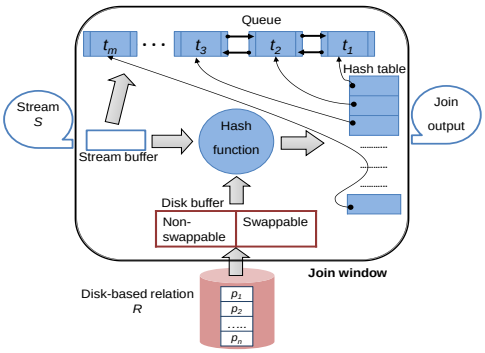


Fig. 2. Architecture of HYBRIDJOIN and X-HYBRIDJOIN. The only difference between the two algorithms is that in X-HYBRIDJOIN the disk buffer is split and its two parts are treated differently, as explained in the text.

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 that stream tuple as an output.

Also, in each iteration, HYBRIDJOIN evicts stream tuples that have been matched. This is justified through the assumption that the join attribute is unique in R. Evicting a tuple means it is deleted from the hash table and the queue. The algorithm also keeps a counter w of the evicted tuples. After processing the whole disk buffer, the algorithm reads w new tuples from the stream buffer, loads them in the hash table along with 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 follows a power law, or Zipfian distribution [11]. The power law is characterized by its exponent. For an exponent < 1 the distribution is said to have a long tail, for an exponent > 1 the distribution has a short tail. For exponent 1 we get the distribution of Zipf's law, which gave rise to the general term Zipfian distribution. In sales, the 80/20 rule is used to model the scenario where the frequency of selling a small number of products is significantly higher compared to the rest of the product, often simplified in the 80/20 rule. The 80/20 rule corresponds to an exponent slightly smaller than 1 [13].

Our aim is to describe an algorithm that takes advantage of the likely distribution of the data. Therefore we created a dataset generator that can create artificial data sets following a power law with an exponent that can be chosen freely. In all our experiments, the master data is sorted with respect to the access frequency.

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 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, 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 divide the disk buffer into two parts. One 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. 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. As a natural default setting, we assign the same amount of memory to both parts.

4.2 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 Equation 2 calculates the processing cost for w tuples. The symbols used to measure the costs are specified in Table 1.

Table 1. Notations used in cost estimation of X-HYBRIDJOIN

Parameter name	**Symbol**
Total allocated memory (*bytes*)	M
Service rate (*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-α
Cost to read one disk partition into the disk buffer (*nanosecs*)	$c_{I/O}(v_P)$
Cost to lookup 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 hash table and the queue (*nanosecs*)	c_A
Total cost for one loop iteration of X-HYBRIDJOIN (*secs*)	c_{loop}

Memory cost In X-HYBRIDJOIN, the disk buffer is divided into two equal parts. One is swappable, the other is non-swappable. As said before, 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 parts= $v_P + v_P = 2v_P$ (in the case of HYBRIDJOIN it is v_P only.)
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.

$$M = 2v_P + \alpha(M - 2v_P) + (1 - \alpha)(M - 2v_P) \quad (1)$$

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).

Processing cost In this section, we calculate the processing cost for the proposed 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. 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 = $d_T.c_H + d_T.c_H = 2d_T.c_H$ (in the case of HYBRIDJOIN it is $d_T.c_H$ only.)

Cost to generate the output for w matching tuples = $w.c_O$
Cost to remove w tuples from the hash table and the queue = $w.c_E$
Cost to read w tuples from stream S into the stream buffer = $w.c_S$
Cost to append w tuples in the hash table and the queue = $w.c_A$
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:

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

For all c_{loop} seconds the algorithm processes w tuples of stream S; therefore, the service rate μ can be calculated by dividing w by the cost for one loop iteration as shown in Equation 3.

$$\mu = \frac{w}{c_{loop}} \quad (3)$$

5 Experiments

We performed experiments to compare the performance of our algorithms with MESHJOIN. We also validate the measured cost by comparing it with the calculated cost for each algorithm. As mentioned before, we use synthetic datasets with a known skew.

5.1 Experimental setup

Hardware specifications: We carried out our experiments on a Pentium-IV 2X2.13GHz machine under WindowsXP. The maximum memory we allocated for our experiments is 250MB. We implemented the algorithm in Java. To measure the memory and processing time, we used built-in plugins provided by Apache and Java API respectively.
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 data set 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.
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 one hour. We started our measurements after 20 minutes and keep measuring for 20 minutes. For added accuracy,

Table 2. Data specification

Parameter	**value**
Disk-based data	
Size of disk-based relation R	0.5 *million* to 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

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 4000 measurements for one setting. Moreover, during the execution of the algorithm no other application was running in parallel.

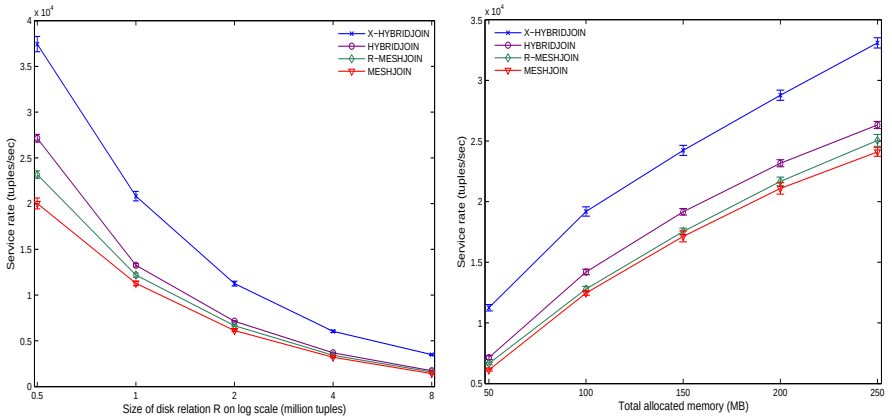
5.2 Experimental results

In our experimental study, we analyzed the results from three different perspectives. Firstly, we compare the performance of HYBRIDJOIN and X-HYBRIDJOIN with the other related algorithms. Secondly, we examine the role of the non-swappable part of the disk buffer in stream processing. Finally, we validate our predicted cost model through experiment.

Performance comparisons The two possible parameters that can vary and directly affect the performance of the algorithms under test are the total available memory for the algorithm and the size of the disk-based relation. In our experiments, we tested the algorithms for different values of these parameters and compared their performance.

Performance comparisons for varying size of the disk-based relation: In the experiment shown in Figure 3(a), we assumed the total allocated memory for the join was fixed while the size of the disk-based relation R was grown exponentially. Figure 3(a) 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.

Performance comparisons when the size of available memory varies: In our second experiment, we analysed the performance of X-HYBRIDJOIN



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

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

Fig. 3. Performance comparisons

using different memory budgets while the size of R is fixed (2 million tuples). Figure 3(b) presents the results of our experiment. The figure indicates that, for all memory budgets, the performance of X-HYBRIDJOIN is again significantly better than 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 that 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.

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 disk buffer. The results of this experiment are shown in Figure 4. As before, we set the size of the non-swappable part to be equal to the size of the swappable part. It is clear from the figure that in 4000 iterations when the memory budget is 50 MB and the size of R is 2 million tuples, about 0.4 million stream tuples are processed through the non-swappable part of the disk buffer and this number increases if we increase the total allocated memory. For 250 MB memory with the same size of R (2 million tuples),

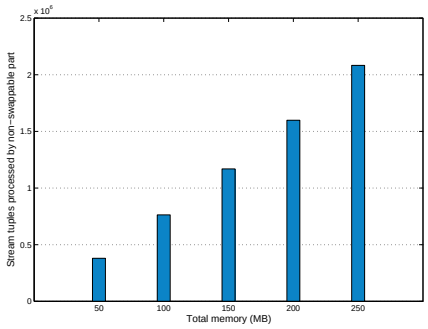


Fig. 4. Total number of stream tuples processed with non-swappable part of disk buffer in *4000* iteration

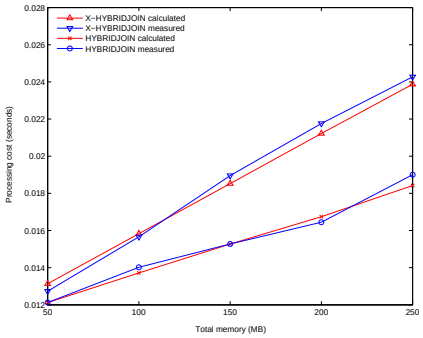


Fig. 5. Cost validation

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.

Cost validation We validate our results by comparing the predicted cost with the measured cost. Figure 5 presents the comparisons of both costs for each algorithm. In the figure, it can be seen that for each algorithm the predicted cost closely matches the measured cost, which is evidence of the consistency of our study.

6 Conclusions and future work

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 modifications over MESHJOIN. The first modification is the use of an index on disk-based master data. The second modification 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. 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 algorithms.

In the future we plan to tune the X-HYBRIDJOIN algorithm in order to utilize the available memory resources optimally.
Source URL: The source of our implementations and pseudo-codes can be downloaded using the given URL:
https://www.cs.auckland.ac.nz/research/groups/serg/src/

References

1. Karakasidis, A., Vassiliadis, P., Pitoura, E.: ETL queues for active data warehousing. In: IQIS '05: Proceedings of the 2nd International Workshop on Information Quality in Information Systems, pp. 28–39. ACM, New York, NY, USA(2005)
2. Naeem, M. A., Dobbie, G., Weber, G.: An Event-Based Near Real-Time Data Integration Architecture. In: Enterprise Distributed Object Computing Conference Workshops, pp. 401-404. IEEE, Munich, Germany(2008)
3. Labio, W., Yang, J., Cui, Y., Garcia-Molina, H., Widom, J.: Performance Issues in Incremental Warehouse Maintenance. In: VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases, pp. 461–472, San Francisco, CA, USA(2000)
4. Labio, W. J., Wiener, J. L., Garcia-Molina, H., Gorelik, V.: Efficient resumption of interrupted warehouse loads. In: SIGMOD Rec., vol. 29, no. 2, pp. 46–57, New York, NY, USA(2000)
5. Nguyen, A., Tjoa, A.: Zero-Latency data warehousing for hetrogeneous data sources and continuous data streams. In: iiWAS'2003 - The Fifth International Conference on Information Integration and Web-based Applications Services, pp. 55–64, Austrian Computer Society(OCG)(2003)
6. Golab, L., Johnson, T., Seidel, J. S., Shkapenyuk, V.: Stream warehousing with DataDepot. In: Proceedings of the 35th SIGMOD International Conference on Management of Data, pp. 847–854, Providence, Rhode Island, USA (2009)
7. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Longman Publishing Co., Boston, MA, USA (2003)
8. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N.E.: Supporting Streaming Updates in an Active Data Warehouse. In: ICDE 2007. IEEE 23rd International Conference on Data Engineering, pp. 476-485. Los Alamitos, CA, USA(2007)
9. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N.: Meshing Streaming Updates with Persistent Data in an Active Data Warehouse. In: IEEE Trans. on Knowl. and Data Eng., vol. 20, no. 7, pp. 976–991, Piscataway, NJ, USA(2008)
10. Chakraborty, A., Singh, A.: A partition-based approach to support streaming updates over persistent data in an active datawarehouse. In: IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–11, IEEE Computer Society, Washington, DC, USA, (2009)
11. Anderson, C.: The Long Tail: Why the Future of Business is Selling Less of More., 2006, Hyperion
12. Naeem, M. A., Dobbie, G., Weber, G.: 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, (2010)
13. Knuth, Donald E.: The art of computer programming., pp. 400-401, Addison-Wiley, Reading, Mass.,(1968)