Tuned X-HYBRIDJOIN for Near-Real-Time Data Warehousing

M. Asif Naeem

School of Computing and Mathematical Sciences, Auckland University of Technology, Private Bag 92006, Auckland 1142, New Zealand
mnaeem@aut.ac.nz

Abstract. Near-real-time data warehousing defines how updates from data sources are combined and transformed for storage in a data warehouse as soon as the updates occur. Since these updates are not in warehouse format, they need to be transformed and a join operator is usually required to implement this transformation. A stream-based algorithm called X-HYBRIDJOIN (Extended Hybrid Join), with a favorable asymptotic runtime behavior, was previously proposed. However, X-HYBRIDJOIN does not tune its components under limited available memory resources and without assigning an optimal division of memory to each join component the performance of the algorithm can be suboptimal. This paper presents a variant of X-HYBRIDJOIN called Tuned X-HYBRIDJOIN. The paper shows that after proper tuning the algorithm performs significantly better than that of the previous X-HYBRIDJOIN, and also better as other join operators proposed for this application found in the literature. The tuning approach has been presented, based on measurement techniques and a revised cost model. The experimental results demonstrate the superior performance of Tuned X-HYBRIDJOIN.

Keywords: Data warehousing, Tuning and performance optimization, Data transformation, Stream-based joins

1 Introduction

Near real-time data warehousing exploits the concepts of data freshness in traditional static data repositories in order to meet the required decision support capabilities. The tools and techniques for promoting these concepts are rapidly evolving. Most data warehouses have already switched from a full refresh [7] to an incremental refresh policy [4]. Further the batch-oriented, incremental refresh approach is moving towards a continuous, incremental refresh approach.

One important research area in the field of data warehousing is data transformation, since the updates coming from the data sources are not in the format required for the data warehouse. Furthermore, a join operator is required to implement the data transformation.

In traditional data warehousing the update tuples are buffered in memory and joined when resources become available [6]. Whereas, in real-time data warehousing these update tuples are joined when they are generated in the data sources. One important factor related to the join is that both inputs of the join come from different sources with different arrival rates. The input from the data sources is in the form of an update stream which is fast, while the access rate of the lookup table is comparatively slow due to disk I/O cost. It creates a bottleneck in processing of update stream and the research challenge here is to minimize this bottleneck by optimizing the performance of the join operator.

To overcome these challenges a novel stream-based join algorithm called X-HYBRIDJOIN (Extended Hybrid Join) [1] was proposed recently by the author. This algorithm not only addressed the issues described above but also was designed to take into account typical market characteristics, commonly known as the 80/20 sale Rule [2]. According to this rule 80 percent of sale focus on only 20 percent of the products, i.e., one observes a Zipfian distribution. To achieve this objective, one component of the algorithm called disk-buffer is divided into two equal parts. The contents of one part of the disk buffer (called the non-swappable part) are kept fixed while the contents of the other part (called the swappable part) are exchanged for each iteration of the algorithm. As the non-swappable part of the disk buffer always contains the most frequently used disk pages, most stream tuples can be processed without invoking the disk. Although the author presented an adaptive algorithm to adopt the typical market characteristics, the components of the algorithm are not tuned to make efficient use of available memory resources. Further details about this issue are provided in Section 3.

On the basis of these observations, a revised X-HYBRIDJOIN is proposed with name Tuned X-HYBRIDJOIN. The cost model of existing X-HYBRIDJOIN is revised and the components of the proposed algorithm are tuned based on that cost model. As a result the available memory is distributed among all components optimally and consequently it improves the performance of the algorithm significantly.

The rest of the paper is structured as follows. The related work to proposed algorithm is presented in Section 2. Section 3 describes problem statement about the current approach. The proposed solution for the stated problem is presented in Section 4. Section 5 presents the tuning of the proposed algorithm based on the revised cost model. The experimental study is discussed in Section 6 and finally Section 7 concludes the paper.

2 Related work

Some techniques have already been introduced to process the join queries over continuous streaming data [5]. This section presents only those approaches which are directly related to the stated problem domain.

A stream-based algorithm called Mesh Join (MESHJOIN) [8] was designed specifically for joining a continuous stream with disk-based data in an active data warehouse. This is an adaptive approach but there are some research issues related to inefficient memory distribution among join components due to unnecessary constraints and an inefficient strategy for accessing the disk based relation.

R-MESHJOIN (reduced Mesh Join) [9] is a revised version of MESHJOIN that focuses on the optimal distribution of memory among the join components. The R-MESHJOIN algorithm introduces the new strategy for memory distribution among the join components by implementing real constraints. However, the mechanism used for accessing the disk based relation is similar to MESHJOIN.

One approach to improve MESHJOIN has been a partition-based join algorithm [10]. It uses a two-level hash table in order to attempt to join stream tuples as soon as they arrive, and uses a partition-bases waiting area for other stream tuples. In this approach the author keeps focus about the analysis of the stream buffer in terms of back log tuples rather than analysing the performance of the algorithm.

A recent piece of work introduces a novel stream-based join called HYBRIDJOIN (Hybrid Join) [11]. The key objective of this effort is to minimize the disk overhead using an index-based approach and to deal with intermittency in the stream. Although both issues are successfully addressed in this approach, it is not optimal with respect to stream data with Zipfian distribution.

3 Preliminaries and problem definition

This section presents a working overview of X-HYBRIDJOIN along with the research issue. In the field of real-time data warehousing X-HYBRIDJOIN is an adaptive algorithm for joining the bursty data stream with disk-based master data. Although the typical characteristics of market data are considered, optimal settings for the available limited memory resources are not considered. Before describing the problem it is first necessary to explain the major components of X-HYBRIDJOIN and the role of each component. Figure 1 presents an abstract level working overview of X-HYBRIDJOIN where m is the number of partitions in the queue to store stream tuples and n is the number of pages in disk-based master data R. Moreover, R is assumed to be sorted with respect to the access frequency. The stream tuples are stored in the hash table while the join attribute values are stored in the queue. The queue is implemented using a doubly linked-list data structure to allow the random deletion of matching tuples. The disk buffer is used to load the disk pages into memory. To make efficient use of R by minimizing the disk access cost, the disk buffer is divided into two equal parts. One is called the non-swappable part which stores a small but most frequently used portion of R into memory on a permanent basis. The other part of the disk buffer is swappable and for each iteration it loads the disk page p_i from R into memory.

Before the join process starts X-HYBRIDJOIN loads the most frequently used page of R into the non-swappable part of the disk buffer. During the join process, for each iteration the algorithm dequeues the oldest join attribute value from the queue and using this value as an index it loads the relevant disk page into the swappable part of the disk buffer. After loading the disk page into memory the algorithm matches each of the disk tuples available in both the swappable and non-swappable parts of the disk buffer with the stream tuples

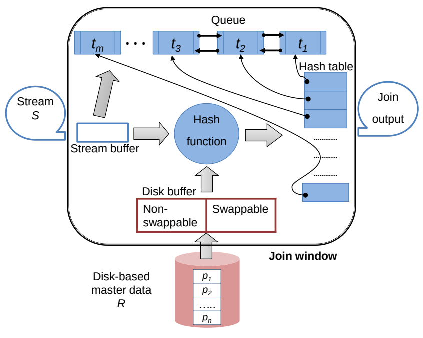


Fig. 1. X-HYBRIDJOIN working overview

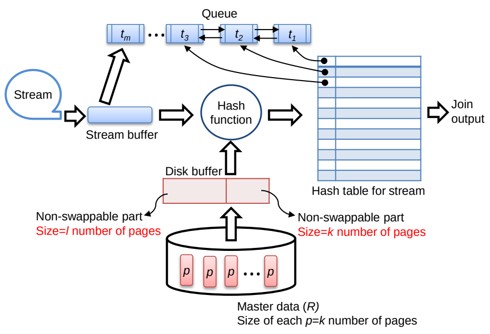


Fig. 2. Memory architecture for Tuned X-HYBRIDJOIN

in the hash table. If the tuple match, the algorithm generates the resulting tuple as output and deletes the stream tuple from the hash table along with its join attribute value from the queue. In the next iteration the algorithm again dequeues the oldest element from the queue, loads the relevant disk page into the swappable part of the disk buffer and repeats the procedure.

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 disk buffer in HYBRIDJOIN [11] 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. The problem considered in this paper is to tune the size of both parts of the disk buffer so that the memory distribution among these two components is optimal. Once these two components acquire optimal settings, based on that memory can be assigned to the rest of join components.

4 Proposed solution

As a solution for the above stated problem, a revised version of X-HYBRIDJOIN called Tuned X-HYBRIDJOIN is proposed. This section presents the memory architecture and a revised cost model for the proposed algorithm. Most importantly, the tuning for the proposed algorithm is presented while the tuning procedure is based on both a measurement strategy and the cost model.

4.1 Memory architecture

The memory architecture that Tuned X-HYBRIDJOIN uses is shown in Figure 2. From the figure, Tuned X-HYBRIDJOIN includes the same number of components as X-HYBRIDJOIN however, the memory size for each component is different to that in X-HYBRIDJOIN. In Tuned X-HYBRIDJOIN since memory is allocated to the each component after executing the tuning module and therefore, each component is assigned an optimal size of memory. Particulary, X-HYBRIDJOIN uses same memory for the both swappable and non-swappable parts of the disk buffer but after tuning it has been explored that the optimal size of memory for the both components is different. The reason for it is presented in Section 5.

4.2 Cost calculation

This section revises the cost formulas derived in X-HYBRIDJOIN. The reason for revising the cost model is that X-HYBRIDJOIN uses equal memory for both the swappable and non-swappable parts of the disk buffer, and therefore the formulas do not apply for other relative sizes. Following the style of cost modeling used for MESHJOIN, the cost for any algorithm is expressed in terms of memory and processing time. Equation 1 describes the total memory used to implement the algorithm while Equation 2 calculates the processing cost for w tuples.

Memory cost Since the optimal values for the sizes of both the swappable part and non-swappable part can be different, it is assumed k number of pages for the swappable part and l number of pages for the non-swappable part. Overall the largest portion of the total memory is used for the hash table while a much smaller amount is used for each of the disk buffer and the queue. The memory for each component can be calculated as given below:
Memory for the swappable part of disk buffer (bytes)= $k \cdot v_P$ (where v_P is the size of each disk page in bytes).
Memory for the non-swappable part of disk buffer (bytes)= $l \cdot v_P$.
Memory for the hash table (bytes)= $\alpha[M - (k + l)v_P]$ (where M is the total allocated memory and α is memory weight for the hash table).
Memory for the queue (bytes)= $(1-\alpha)[M-(k+l)v_P]$ (where $(1-\alpha)$ is memory weight for the queue).
The total memory used by the algorithm can be determined by aggregating the above.

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

Currently the memory reserved for the stream buffer is not included because it is small (0.05 MB was sufficient in all experiments presented in this paper).

Processing cost This section presents the processing cost for the proposed approach. The cost for one iteration of the algorithm is denoted by c_{loop} and express it as the sum of the costs for the individual operations. Therefore the processing cost for each component is first calculated separately.
Cost to read the non-swappable part of disk buffer (nanoseconds) = $c_{I/O}(l \cdot v_P)$.
Cost to read the swappable part of disk buffer (nanoseconds)= $c_{I/O}(k \cdot v_P)$.

Cost to look-up the non-swappable part of 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 disk buffer in terms of tuples, v_P is size of disk page in bytes, v_R is size of disk tuple in bytes, and c_H is look-up cost for one disk tuple in the hash table).
Cost to look-up the swappable part of 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 disk buffer in terms of tuples).
Cost to generate the output for w matching tuples (nanoseconds) $= w \cdot c_O$ (where c_O is cost to generate one tuple as an output).
Cost to delete w tuples from the hash table and the queue (nanoseconds)$= w \cdot c_E$ (where c_E is cost to remove one tuple from the hash table and the queue).
Cost to read w tuples from stream S into the stream buffer (nanoseconds)$= w \cdot c_S$ (where c_S is cost to read one stream tuple into the stream buffer).
Cost to append w tuples into the hash table and the queue (nanoseconds)$= w \cdot c_A$ (where c_A is cost to append one stream tuple in the hash table and the queue).
As the non-swappable part of the disk buffer is read only once before execution starts, it is excluded. The total cost for one loop iteration is:

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

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

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

5 Tuning

The stream-based join operators normally execute within limited memory and therefore tuning of join components is important to make efficient use of the available memory. For each component in isolation, more memory would be better but assuming a fixed memory allocation there is a trade-off in the distribution of memory. Assigning more memory to one component means less memory for other components. Therefore it needs to find the optimal distribution of memory among all components in order to attain maximum performance. A very important component is the disk buffer because reading data from disk to memory is expensive.

In the proposed approach tuning is first performed through performance measurements by considering a series of values for the sizes of the swappable and non-swappable parts of the disk buffer. Later a mathematical model for tuning is also derived from the cost model. Finally, the tuning results of both approaches are compared to validate the cost model. The details about the experimental setup are presented in Table 1.

5.1 Tuning through measurements

This section presents the tuning of the key components of the algorithm through measurements. In the measurement approach the performance is tested on par-

Table 1. Data specification

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

ticular memory settings for swappable and non-swappable parts rather than on every contiguous value.

The measurement approach assumes the size of total memory and the size of R are fixed. The sizes for the swappable and non-swappable parts vary in such a way that for each size of the swappable part the performance is measured against a range of sizes for the non-swappable part. By changing the sizes for both parts of the disk buffer the memory sizes for the hash table and the queue are also affected.

The performance measurements for varying the sizes of both swappable and non-swappable parts are shown in Figure 3. The figure shows that the performance increases rapidly by increasing the size for the non-swappable part. After reaching a particular value for the size of 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. On the other hand, due to higher look-up cost and the fact that less memory is available for the hash table the performance decreases gradually. A similar behavior is seen when the performance against the swappable part is tested. In this case, after attaining the maximum performance it decreases rapidly because of an increase in the I/O cost for loading the growing swappable part. From the measurements shown in the figure it is possible to approximate the optimal settings for both the swappable and non-swappable parts by finding the maximum on the two-dimensional surface.

5.2 Tuning based on cost model

A mathematical model for the tuning is also derived based on the cost model presented in Section 4.2. From Equation 3 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 main components on which the value

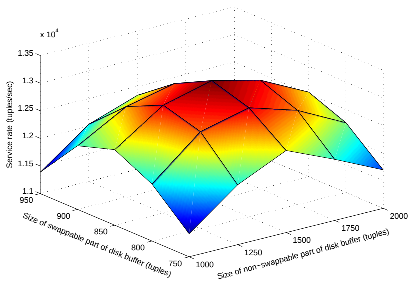


Fig. 3. Tuning using measurement approach

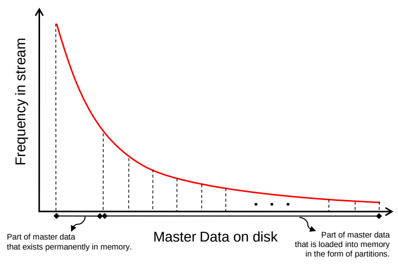

Frequency in stream
Master Data on disk
Part of master data that exists permanently in memory.
Part of master data that is loaded into memory in the form of partitions.

Fig. 4. A sketch of matching probability of R in stream

of w depends are:- size of the non-swappable part (d_N), size of the swappable part (d_S), size of the master data (R_t), and size of the hash table (h_S).

Typically the stream of updates can be approximated through 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 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]. 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 master data 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 every next 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 Figure 4, 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 one moves toward the end position in R.

The size of the hash table is another component that affects w. The reason is simple: if there are more stream tuples in memory, the number of matches will be greater and vice versa. Before driving the formula to calculate w it is first necessary to understand the working strategy of Tuned X-HYBRIDJOIN. Consider for a moment that the queue contains stream tuples instead of just join attribute values. Tuned 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 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 a 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 as well as the swappable parts of R. Therefore, to calculate w it is first needed to calculate both these probabilities.

Therefore, if the stream of updates S obeys Zipf's law, then the matching probability for any swappable partition k with the old part of the queue can be determined mathematically as shown below.

$$p_k = \frac{\sum\limits_{x=d_N+(k-1)d_S+1}^{d_N+kd_S} \frac{1}{x}}{\sum\limits_{x=d_N+1}^{R_t} \frac{1}{x}}$$

Each summation in the above equation generates a harmonic series, which can be summed up using the formula $\sum\limits_{x=1}^{k} \frac{1}{x} = \ln k + \gamma + \varepsilon_k$, where γ is a Euler's constant whose value is approximately equal to 0.5772156649 and ε_k is another constant which is $\approx \frac{1}{2k}$. The value of ε_k approaches 0 as k goes to ∞ [3]. In this paper the value of $\frac{1}{2k}$ 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 Equation 4.

$$\overline{p}_{old} = \frac{\sum\limits_{k=1}^{n} p_k}{n} = \frac{1}{n} \tag{4}$$

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 Equation 5.

$$\overline{p}_{new} = p_N + \frac{1}{n} p_S \tag{5}$$

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.

$$p_N = \frac{\sum_{x=1}^{d_N} \frac{1}{x}}{\sum_{x=1}^{R_t} \frac{1}{x}} \qquad \text{and} \qquad p_S = \frac{\sum_{x=d_N+1}^{R_t} \frac{1}{x}}{\sum_{x=1}^{R_t} \frac{1}{x}}$$

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 are the average number of matches per outer iteration with both the swappable and non-swappable parts, then w can be calculated by applying the binomial probability distribution on Equations 4 and 5 as given below.

$$w = (h_S - w)\overline{p}_{old}(1 - \overline{p}_{old}) + w\overline{p}_{new}(1 - \overline{p}_{new})$$

After simplification the final formula to calculate w is described in Equation 6.

$$w = \frac{h_S\overline{p}_{old}(1 - \overline{p}_{old})}{1 + \overline{p}_{old}(1 - \overline{p}_{old}) - \overline{p}_{new}(1 - \overline{p}_{new})} \tag{6}$$

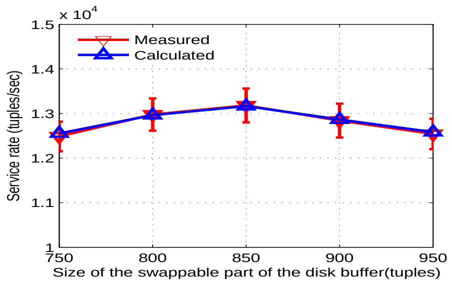
By using the values of w and c_{loop} in Equation 3 the algorithm can be tuned.

5.3 Comparisons of both approaches

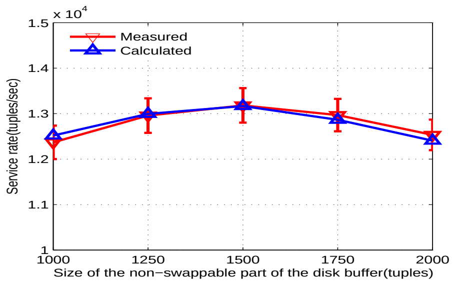
To validate the cost model the tuning results based on the measurement approach are compared with those that are achieved through cost model.
Swappable part: This experiment compares the tuning results for the swappable part of the disk buffer using both the measurement and cost model approaches. The tuning results of each approach (with 95% confidence interval in case of measurement approach) are shown in Figure 5 (a). From the figure it is evident that at every position the results in both cases are similar, with only 0.5% deviation.
Non-swappable part: Similar to before, the tuning results of both approaches for the non-swappable part of the disk buffer are also compared. The results are shown in Figure 5 (b). Again, it can be seen from the figure, the results in both cases are nearly equal with a deviation of only 0.6%. This proves the correctness of the tuning module.

6 Experimental study

To strengthen the arguments an experimental evaluation of proposed Tuned X-HYBRIDJOIN is performed using the synthetic datasets. Normally, in Tuned X-HYBRIDJOIN kinds of algorithms, the total memory and the size of R are the common parameters that vary frequently. Therefore, the experiments presented here compare the performance by varying both parameters individually.
Performance comparisons for different sizes of R: This experiment compares the performance of Tuned X-HYBRIDJOIN with the other related algorithms for different sizes of R. Therefore it is assumed that the size of R varies

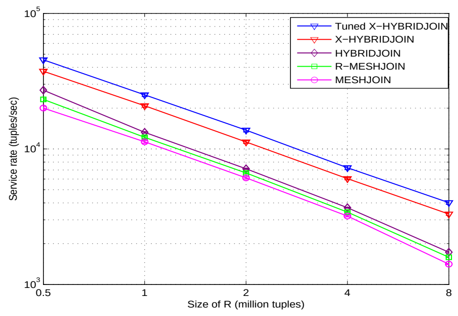


(a) Tuning Comparison for swappable part: based on measurements Vs based on cost model

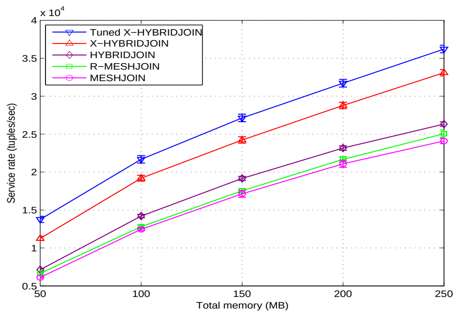


(b) Tuning Comparison for non-swappable part: based on measurements Vs based on cost model

Fig. 5. Comparisons of tuning results



(a) Size of disk-based relation varies (on log-log scale)



(b) Total allocated memory varies

Fig. 6. Performance comparisons

exponentially while the total memory budget remains fixed (50MB) for all values of R. For each value of R the performance is measured separately. The performance results of this experiment are shown in Figure 6 (a). From the figure it is clear that for all settings of R the Tuned X-HYBRIDJOIN performs significantly better than other approaches.

Performance comparisons for different memory budgets: Second experiment analyses the performance of Tuned X-HYBRIDJOIN using different memory budgets while the size of R is fixed (2 million tuples). Figure 6 (b) depicts the performance results. From the figure it can be observed that for all memory budgets the Tuned X-HYBRIDJOIN again performs significantly better than all approaches. This improvement increases gradually as the total memory budget increases.

7 Conclusions

This paper investigates a well known stream-based join algorithm called X-HYBRIDJOIN. Main observation about X-HYBRIDJOIN is that the tuning factor is not considered but it is necessary, particularly when limited memory resources are available to execute the join operation. By omitting the tuning factor, the available memory cannot be distributed optimally among the join components and consequently the algorithm cannot perform optimally. This paper presents a variant version of X-HYBRIDJOIN called Tuned X-HYBRIDJOIN. The cost model presented in X-HYBRIDJOIN is revised and the proposed algorithm is tuned based on that revised cost model. To strengthen the arguments a prototype of Tuned X-HYBRIDJOIN is implemented and the performance with existing approaches is compared.

References

1. Naeem, M. A., Dobbie, G., Weber, G.: X-HYBRIDJOIN for Near-real-time Data Warehousing. In BNCOD'11: Proceedings of 28th British National Conference on Databases, pp. 33–47, Springer-Verlag, Berlin Heidelberg, (2011)
2. Anderson, C.: The Long Tail: Why the Future of Business is Selling Less of More., 2006, Hyperion
3. Milton A., Irene A. S.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables., 1964, Ninth Dover printing, Tenth GPO printing, New York.
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. Golab, L., Tamer Özsu, M.: Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In: VLDB'03, pp. 500–511, Berlin, Germany, (2003)
6. Wilschut, A. N., Apers, P. M. G: Dataflow query execution in a parallel main-memory environment. In: Distrib. Parallel Databases., vol. 1, no. 1, pp. 103–128, Hingham, MA, USA, (1993)
7. Gupta, A., Mumick, I. S.: Maintenance of Materialized Views: Problems, Techniques, and Applications. In: IEEE Data Engineering Bulletin, vol. 18, pp. 3–18, (2000)
8. 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)
9. 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)
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. Naeem, M. A., Dobbie, G., Weber, G.: HYBRIDJOIN for Near-real-time Data Warehousing. In: International Journal of Data Warehousing and Mining (IJDWM), vol. 7, no.4, IGI-Global, (2011)