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

brought to you by CORE

provided by AIS Electronic Library (AISeL)

Association for Information Systems

AIS Electronic Library (AISeL)

CONF-IRM 2019 Proceedings

International Conference on Information Resources Management (CONF-IRM)

5-2019

Optimising HYBRIDJOIN to Process Semi-Stream Data in Near-real-time Data Warehousing

M Asif Naeem
Auckland University of Technology, mnaeem@aut.ac.nz

Omer Aziz
NFC Institute of Engineering & Technology, omer.aziz@nfciet.edu.pk

Noreen Jamil
Unitec Institute of Technology, njamil@unitec.ac.nz

Follow this and additional works at: https://aisel.aisnet.org/confirm2019

Recommended Citation

Naeem, M Asif; Aziz, Omer; and Jamil, Noreen, "Optimising HYBRIDJOIN to Process Semi-Stream Data in Near-real-time Data Warehousing" (2019). *CONF-IRM 2019 Proceedings*. 27.
https://aisel.aisnet.org/confirm2019/27

This material is brought to you by the International Conference on Information Resources Management (CONF-IRM) at AIS Electronic Library (AISeL). It has been accepted for inclusion in CONF-IRM 2019 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

Optimising HYBRIDJOIN to Process Semi-Stream Data in Near-real-time Data Warehousing

M. Asif Naeem
Auckland University of Technology,
Auckland, New Zealand
mnaeem@aut.ac.nz

Omer Aziz
NFC Institute of Engineering &
Technology, Multan, Pakistan
omer.aziz@nfciet.edu.pk

Noreen Jamil
Unitec Institute of Technology,
Auckland, New Zealand
njamil@unitec.ac.nz

Abstract

Near-real-time data warehousing plays an essential role for decision making in organizations where latest data is to be fed from various data sources on near-real-time basis. The stream of sales data coming from data sources needs to be transformed to the data warehouse format using disk-based master data. This transformation process is a challenging task due to slow disk access rate as compare to the fast stream data. For this purpose, an adaptive semi-stream join algorithm called HYBRIDJOIN (Hybrid Join) is presented in the literature. The algorithm uses a single buffer to load partitions from the master data. Therefore, the algorithm has to wait until the next disk partition overwrites the existing partition in the buffer. As the cost of loading the disk partition into the buffer is a major cost in the total algorithm's processing cost, this leaves the performance of the algorithm sub-optimal. This paper presents optimisation of existing HYBRIDJOIN by introducing another buffer. This enables the algorithm to load the second buffer while the first one is under join execution. This reduces the time that the algorithm wait for loading of master data partition and consequently, this improves the performance of the algorithm significantly.

Keywords

Real-time Data Warehousing; Semi-Stream Joins; Stream data, Master data

1. Introduction

The industry is moving towards near-real-time data warehouse to make rapid decisions. The tools and techniques for promoting these concepts are rapidly evolving (Golfarelli & Rizzi, 2009; Thomsen & Pedersen, 2005; Vassiliadis, 2009). The batch-oriented, incremental refresh approach is moving towards a continuous, incremental refresh approach (Karakasidis, Vassiliadis, & Pitoura, 2005; Slavin, 1980; Thiele, Fischer, & Lehner, 2009; Tho & Tjoa, 2004).

One important research area in the field of data warehousing is data transformation. Since data generated from business transactions – also called sales data or source data – is not in the format required by near-real-time data warehouse, it needs to be transformed online using disk-based master data before loading it into the data warehouse. Usually this transformation of data is performed using ETL (Extraction Transformation Loading) tools. Common examples of transformations are unit conversion, removal of duplicate tuples, information enrichment, filtering of unnecessary data, sorting of tuples, and translation of source data key.

Let us consider an example for the transformation phase shown in Figure 1 that implements one of the above features, called enrichment. In the example we consider the source data with attributes *product_id*, *qty*, and *date* that are extracted from data sources. At the transformation

layer, in addition to key replacement (from source key *product_id* to warehouse key *s_key*) some other information namely sales price denoted by *s_price* to calculate the total amount and the *vendor* information is also added. In the figure this information with attributes name *s_key*, *s_price*, and *vendor* are extracted at run time from the master data and are used to enrich the source updates using a join operator.

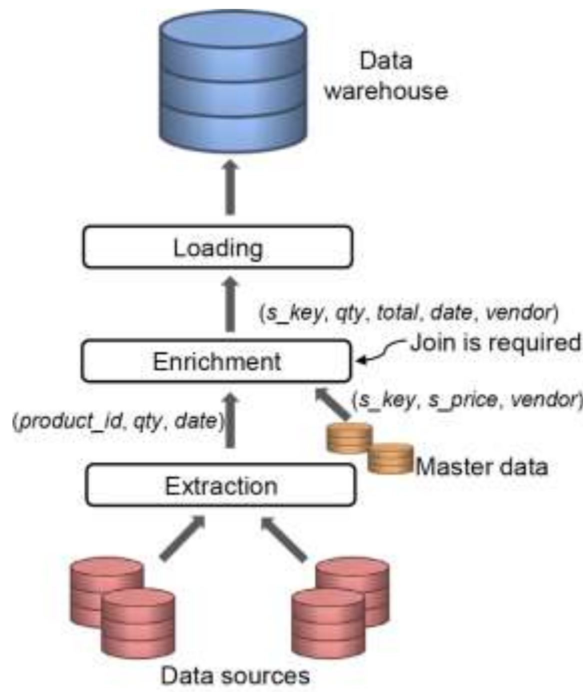


Figure 1: An example of transformation (Naeem, Dobbie, & Weber, 2012).

In traditional data warehousing the source updates are buffered and the join is performed offline. On the other hand, in near-real-time data warehousing this operation needs to be performed as soon as the data is received from the data sources. In implementing the online execution of join, one important challenge is the different arrival rate of both inputs. The stream input is fast and huge in volume while the disk input is slow. The challenge here is to amortise the disk access cost over the fast input stream.

HYBRIDJOIN (Hybrid Join) is one of the optimal semi-stream join algorithms presented in the literature to join streaming data with the master data (Naeem, Dobbie, & Weber, 2011a). The algorithm shows better performance as compared to other hash join algorithms like Mesh Join (N Polyzotis, Skiadopoulos, Vassiliadis, Simitsis, & Frantzell, 2007; Neoklis Polyzotis, Skiadopoulos, Vassiliadis, Simitsis, & Frantzell, 2008), Indexed Nested Loop Join (Ramakrishnan & Gehrke, 2000), and Semi-Stream Index Join (Bornea, Deligiannakis, Kotidis, & Vassalos, 2011). The key objective of this algorithm is to amortize the fast input stream with the slow disk access within limited memory budget and to deal with the bursty nature of the input data stream. The algorithm reads the input stream in chunks and to amortize the fast stream, it keeps a number of chunks in memory at the same time. The number of chunks can be differentiated with respect to the loading time. To organize the order and to keep the record of non-matching stream tuples, the algorithm stores the join attribute values in a component called queue while storing the tuples themselves in a hash table. The algorithm, for each iteration, loads a disk partition from the disk-based master data *R* into memory called disk buffer using oldest value from the queue as an index for the master data access.

Once the disk partition is loaded into memory, the algorithm matches the disk tuples with all available stream tuples in memory. This process is called probing phase. In the case of a match, the algorithm generates a new tuple as an output and deletes the tuple from the hash table along with its join attribute value from the queue. In the next iteration, the algorithm again accesses the oldest value from the queue and repeats the entire procedure.

Although the HYBRIDJOIN algorithm efficiently amortizes the fast input stream using an index-based approach to access *R*, HYBRIDJOIN uses single disk buffer to load partitions of *R*. Thus, in the probing phase when the algorithm processes all the tuples from the disk buffer the algorithm needs to wait until the disk buffer gets refill. This doesn't utilise CPU and memory resources completely. In order to resolve this issue, in this paper we introduce an improved version of HYBRIDJOIN called Optimised HYBRIDJOIN. The new algorithm implements two disk buffers to load the partitions from *R*. These buffers are loaded alternatively for example when first buffer is under the use of join operation – i.e. probing phase – the algorithm loaded the second buffer. In this case the algorithm's join operation does not need to wait for loading the disk buffer which fully exploits CPU and memory resources. Our results of Optimised HYBRIDJOIN show a significant performance improvement as compared to HYBRIDJOIN.

The rest of this paper is structured as follows. The related work is presented in Section 2. Section 3 describes existing HYBRIDJOIN and highlights our observations about the current approach. In Section 4 we present our Optimised HYBRIDJOIN including its architecture, algorithm, and cost model. The experimental study is conducted in Section 5 and finally Section 6 concludes the paper.

2. Related Work

Performance optimization of the semi-stream join process for user-update in near-real-time fashion has been of prime importance for the data warehouse and database research community. A number of approaches have been published in the literature to optimize the performance of semi-stream join operation. In this section, we present the most relevant from these approaches along with their limitations.

The Index Nested Loop Join (INLJ) (Ramakrishnan & Gehrke, 2000) was the first index-based algorithm introduced for joining two static tables. Although the algorithm can be extended to semi-stream join scenario, the performance is very low as the algorithm process only one stream tuple against each disk access of *R*. This creates bottleneck for the streaming data.

The MESHJOIN (Mesh Join) algorithm (N Polyzotis et al., 2007; Neoklis Polyzotis et al., 2008) was introduced with the objective to amortise the slow disk access with as many stream tuples as possible. To perform the join, the algorithm keeps a number of chunks of stream data in memory at the same time. In each iteration the algorithm loads a disk partition into memory and performs the join with all these chunks of stream data. The algorithm performs tuning for efficient memory distribution among the join components, but in the past, we identified some issues related to the access of *R*. Also, MESHJOIN cannot efficiently deal with the characteristic of skew in the stream data.

R-MESHJOIN (Reduced Mesh Join) – an enhanced form of MESHJOIN – was introduced for the better performance (Naeem, Dobbie, Weber, & Alam, 2010). It overcomes the issue of suboptimal memory distribution of MESHJOIN by introducing a new memory architecture. However, R-MESHJOIN implements the same strategy as the MESHJOIN algorithm for accessing *R*.

A partition-based approach was introduced to deal with intermittency in the stream (Chakraborty & Singh, 2009, 2010). It uses a two-level hash table to attempt to join stream tuples as soon as they arrive, and uses a partition-based waiting area for the other stream tuples. The authors did not provide a cost model for their approach. In addition, the algorithm requires a clustered index or an equivalent sorting on the join attribute and it does not prevent starvation of stream tuples.

Semi-Streaming Index Join (SSIJ) (Bornea et al., 2011) was another recent attempt to join S with R. In general, the algorithm is divided into three phases: the pending phase, the online phase and the join phase. In the pending phase, the stream tuples are collected in an input buffer until either the buffer is larger than a predefined threshold or the stream ends. In the online phase, stream tuples from the input buffer are looked up in cached disk blocks. If the required disk tuple exists in the cache, the join is executed. Otherwise, the algorithm flushes the stream tuple into a stream buffer. When the stream buffer is full, the join phase starts where R partitions are loaded from disk using an index and joined until the stream buffer is empty. This means that as partitions are loaded and joined, the join becomes more and more inefficient: partitions that are joined later can potentially join only with fewer tuples because the stream buffer is not refilled between partition loads. By keeping the stream buffer full and selecting lookup elements carefully the performance could be improved.

This problem of semi-stream join was addressed in another approach called X-HYBRIDJOIN (Extended HYBRIDJOIN) (Naeem, Dobbie, & Weber, 2011b). X-HYBRIDJOIN produced the significant performance improvement by keeping the frequent tuples of stream into the non-swappable part of the memory buffer and new records are loaded in to the swappable part of the memory buffer. The algorithm was designed particularly to cope with Zipfian distributions. Although this is an adaptive algorithm and performs better than other similar approaches, the access of the disk-based master data is still a bottleneck in it that needs to be explored further.

The algorithm for joining stream data with a disk-based relation by Derakhshan et al. (Derakhshan, Sattar, & Stantic, 2013) introduced a cache component to store frequent master data tuples and a waiting queue for stream tuples that are not joined through the cache. The algorithm processes this waiting queue in batches. The primary focus of the algorithm was to preserve the arrival order of the stream tuples in the output. While useful for some applications, this order is not important in most applications such as the ones we consider. Preserving this order can cause delays in producing outputs for some stream tuples, e.g. a stream tuple already processed through the cache cannot be output if its predecessor tuple is still in the waiting queue. Derakhshan and others also demonstrated the role of stream-relation joins in a federated stream processing system called MaxStream (Botan et al., 2010).

One recent algorithm, HYBRIDJOIN (Hybrid Join) (Naeem et al., 2011a) addressed the issue of accessing the disk-based relation. An effective strategy to access the disk-based relation was introduced in HYBRIDJOIN. Another advantage is that the algorithm can deal with bursty streams, which is a limitation of both MESHJOIN and R-MESHJOIN. However, the algorithm uses the same suboptimal strategy as in X-HYBRIDJOIN to access the master data and we address that issue in this paper.

3. HYBRIDJOIN and Problem Definition

Figure 2 presents the memory architecture for HYBRIDJOIN (Hybrid Join). In HYBRIDJOIN, memory is divided into several components. The queue Q component implementing double link list data structure is used to store the join attribute values from the stream data. The other component is a hash table H that stores stream tuples. The size of H is hs in terms of

tuples. The disk buffer *db* is an important component which holds the master data partition into memory. The stream buffer is used to temporary hold the incoming stream data for a while until the stream data in memory get processed. HYBRIDJOIN uses index-based approach to access *R*. Therefore, each access of *R* processes at least one tuple of stream data. Due to this fact HYBRIDJOIN gets preference over MESHJOIN. In MESHJOIN it is possible that the algorithm processes no stream tuple against an access of *R*. However, the HYBRIDJOIN algorithm has its own limitations. The algorithm consists of two major phases, loading phase and probing phase, while these phases are executed sequentially. In loading phase, the algorithm loads a partition of *R* to *db*, while in probing phase, the join is performed between *db* and the streaming data stored in *H*. The algorithm has a high I/O cost to retrieve *R* and the probing phase needs to wait while the algorithm remains busy in its loading phase. In this paper we address this limitation that consequently improves the performance of HYBRIDJOIN.

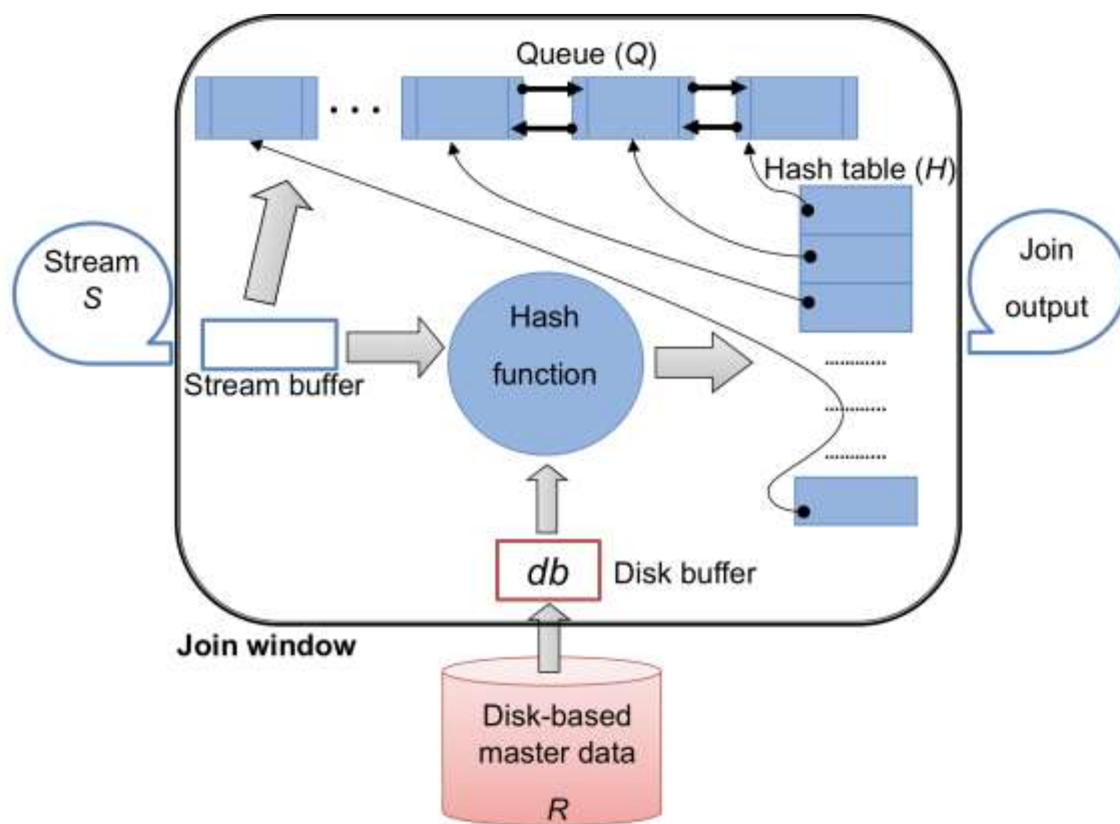


Figure 2: HYBRIDJOIN architecture (Naeem et al., 2013a).

4. Optimised HYBRIDJOIN

As a solution to the problem stated above, we present an optimised version of HYBRIDJOIN called Optimised HYBRIDJOIN. The new algorithm overcomes the high disk I/O by introducing a new disk buffer (*db2*) which allows the parallel execution of two phases – the probing phase and the loading phase – of the existing HYBRIDJOIN algorithm. Figure 3 presents the memory architecture for Optimised HYBRIDJOIN. During the probing phase when the algorithm is using *db1* for the join operation, the algorithm starts loading a partition of *R* into *db2* using index value from *Q*. As soon as the algorithm finishes its probing phase using disk *db1*,

db2 becomes ready for the join operation. Similarly, while the probing phase uses *db2*, *db1* gets ready by loading the next partition of *R*. Thus, the algorithm for every of its iteration doesn't have to wait for the disk buffer loading. In our experiments we observed that this parallel execution of the both phases improves the performance of the algorithm significantly.

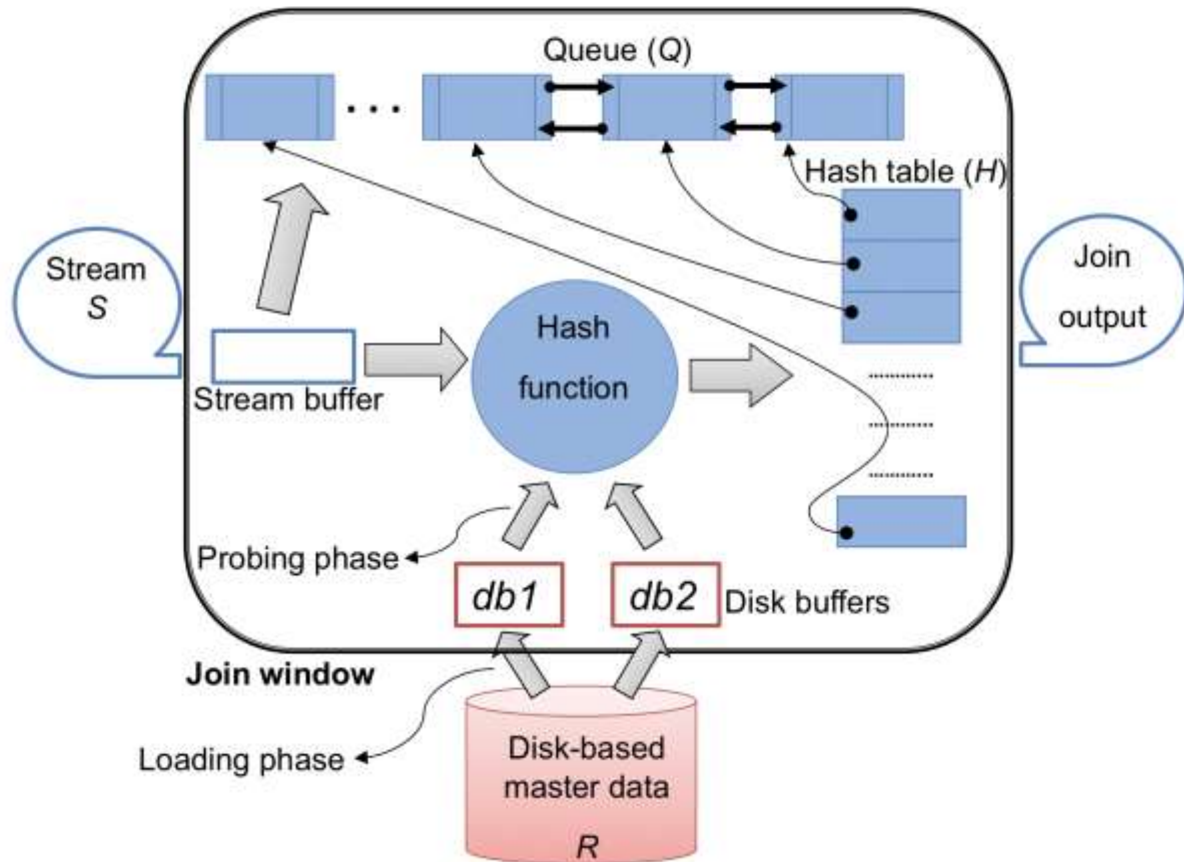


Figure 3: Memory Architecture for Optimised HYBRIDJOIN

4.1 Algorithm

Figures 4 presents procedures for the both probing and loading phase of the new algorithm. First three lines in probing phase – named *ProbingPhase* – presents the input, output and parameters required by the algorithm. Initially since *H* is empty, *w* which is a stream input size for each iteration of the algorithm is equal to the size of the hash table *hs* (line 1). The first loop in the algorithm runs infinitely, which is normal in these type of algorithms (line 2). Inside the first loop the algorithm first checks the availability of the stream data in the stream buffer. If the stream is available the algorithm loads *w* tuples from the stream buffer to *H* along with storing their join attribute values in *Q*. Also, the algorithm reset *w* to zero (lines 3 to 6). Once the stream data has been loaded, the algorithm checks the status of *db1* and *db2*. If *db1* is empty – which means *db2* is ready – the algorithm calls procedure *LoadingPhase* to load a partition of *R* into *db1*. Meanwhile, the algorithm executes its probing phase and performs the join operation between *H* and *db2*. The algorithm generates output for the matched tuples and delete these tuples from *H* and *Q*. This creates empty slots in *H* so algorithm adds these number of empty slots to *w*. Once all tuples from *db2* get processed, the algorithm set *db2* to empty (lines 7 to 16). The algorithm executes the similar steps for *db1* (lines 18 to 27).

Procedure *LoadingPhase* – which is called from the main procedure *ProbingPhase* – receives the empty disk buffer and index value from *Q* in parameters *db* and *i* respectively. The procedure reads a partition of *R* using index value *i* and load that partition on to *db* (lines 1 and 2). The procedure then returns *db* to the main procedure *ProbingPhase* (line 3).

```
PROCEDURE OptimisedHYBRIDJOIN.ProbingPhase()
Input: A master data R with an index on join attribute and Stream of updates S
Output: S ⋈ R
Parameters: w records of S and a partition of R

 1: w ← h_s
 2: while true do
 3:   if stream available then
 4:     Read w records from the stream buffer and load them in H with their join attribute values
        in Q
 5:     w ← 0
 6:   end if
 7:   if db1 is empty then
 8:     db1 ← PROCEDURE OptimisedHYBRIDJOIN.LoadingPhase(db1, Q.index)
 9:     for each record r in db2 do
10:       if r ∈ H then
11:         Output r ⋈ H
12:         Delete matching records from H and Q
13:         w = w + number of matching records in H
14:       end if
15:     end for
16:     Empty db2
17:   else
18:     db2 ← PROCEDURE OptimisedHYBRIDJOIN.LoadingPhase(db2,Q.index)
19:     for each record r in db1 do
20:       if r ∈ H then
21:         Output r ⋈ H
22:         Delete matching records from H and Q
23:         w = w + no of matching records in H
24:       end if
25:     end for
26:     Empty db1
27:   end if
28: end while
29: END PROCEDURE
```

```
PROCEDURE OptimisedHYBRIDJOIN.LoadingPhase(db, i)
Parameters: db and value of index i in Q

1: READ a partition of R using index i
2: LOAD the partition into db
3: RETURN db
4: END PROCEDURE
```

Figure 4: Procedures for probing and loading phases of Optimised HYBRIDJOIN

4.2 Cost model

Memory cost: In Optimised HYBRIDJOIN, the maximum portion of the total memory is used for *H* while a much smaller as compare to *H* is used for *db1*, *db2*, and *Q*. In the following we first calculate the memory for each component separately and then aggregate all these to know how much memory the algorithm consumes in total.

Memory reserved for *db1* and *db2* (bytes) = $2v_P$
Memory reserved for H (bytes) =$\alpha(M - 2v_P)$
Memory reserved for Q (bytes) = $(1 - \alpha)(M - 2v_P)$

Where α and $(1 - \alpha)$ are memory weights for H and Q respectively. The total memory used by HYBRIDJOIN can be determined by aggregating the above all.

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

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

Processing cost: In this section we calculate the processing cost for our Optimised HYBRIDJOIN (Hybrid Join). To make it simple we first calculate the processing cost for one *loop* iteration. In order to calculate the cost for one *loop* iteration the major components are:
Cost to read two disk partitions in *db1* or *db2* = $c_{I/O}(v_P)$
Cost to probe one disk partition into *H*= $\frac{v_P}{v_R} c_H$
Cost to generate the output for *w* matching tuples = wc_O
Cost to delete *w* tuples from *H* and *Q* = wc_E
Cost to read *w* tuples from stream *S* = wc_S
Cost to append *w* tuples into *H* and *Q*= wc_A
Cost to switch current buffer state =$d_B c_C$
By aggregation, the total cost of one loop iterations is:

$$c_{loop} = 10^9 \left[c_{I/O}(v_P) + \frac{v_P}{v_R} c_H + w(c_O + c_E + c_S + c_A) + d_B c_C\right] \quad (2)$$

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

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

5. Experimentation

In this section we performed an experimental evaluation of Optimised HYBRIDJOIN using synthetic but on real pattern skewed datasets. Before presenting our results we first illustrate the environment we used to conduct the experiments.

5.1 Experimental Setup

In order to implement the prototype for both HYBRIDJOIN and Optimised HYBRIDJOIN we used the following hardware and data specifications.

Hardware specification: The experiments were performed on Core i7 with 16GB main memory and 160 GB hard drive. The maximum memory of 250MB was allocated to each of the algorithm.

Data specification: Both of the algorithms were tested with various sizes of *R* which were 0.5 million, 1 million, 2 million, 4 million, and 8 million tuples. We also used different settings for available memory which were 50MB, 100MB, 150MB, 200MB and 250MB. The relation *R* was stored on disk using MySQL database. On real pattern skewed stream data was generated at run time using the same benchmark that was used in HYBRIDJOIN algorithm (Naeem et al., 2011a). The size of each stream tuple was 20 bytes while the size of each tuple of *R* was 120 bytes.
We implemented the experiment in Java using Eclipse IDE. We also used built-in plugins, provided by Apache, and *nanoTime()*, provided by the Java API, to measure the memory and processing time respectively. Currently Optimised HYBRIDJOIN supports join for one-to-one and one-to-many relationships. In order to implement the join for one-to-many relationship it needs to store multiple values in hash table against one key value. However, the hash table provided by Java API does not support this feature therefore, we used Multi-Hash-Map, provided by Apache, as a hash table in our experiments. The hash table has fudge factor of 8.

Measurement strategy: The performance of the algorithm was measured by analysing the number of records processed in unit second. This was called the service rate. The results measurement was started after executing a few loop iterations. For more accuracy we took three readings for each test and considered the third measurement as final. We also calculated confidence interval for every result by considering 95% accuracy. Moreover, during the execution of the algorithm no other application was assumed to run in parallel.

5.2 Performance Evaluation

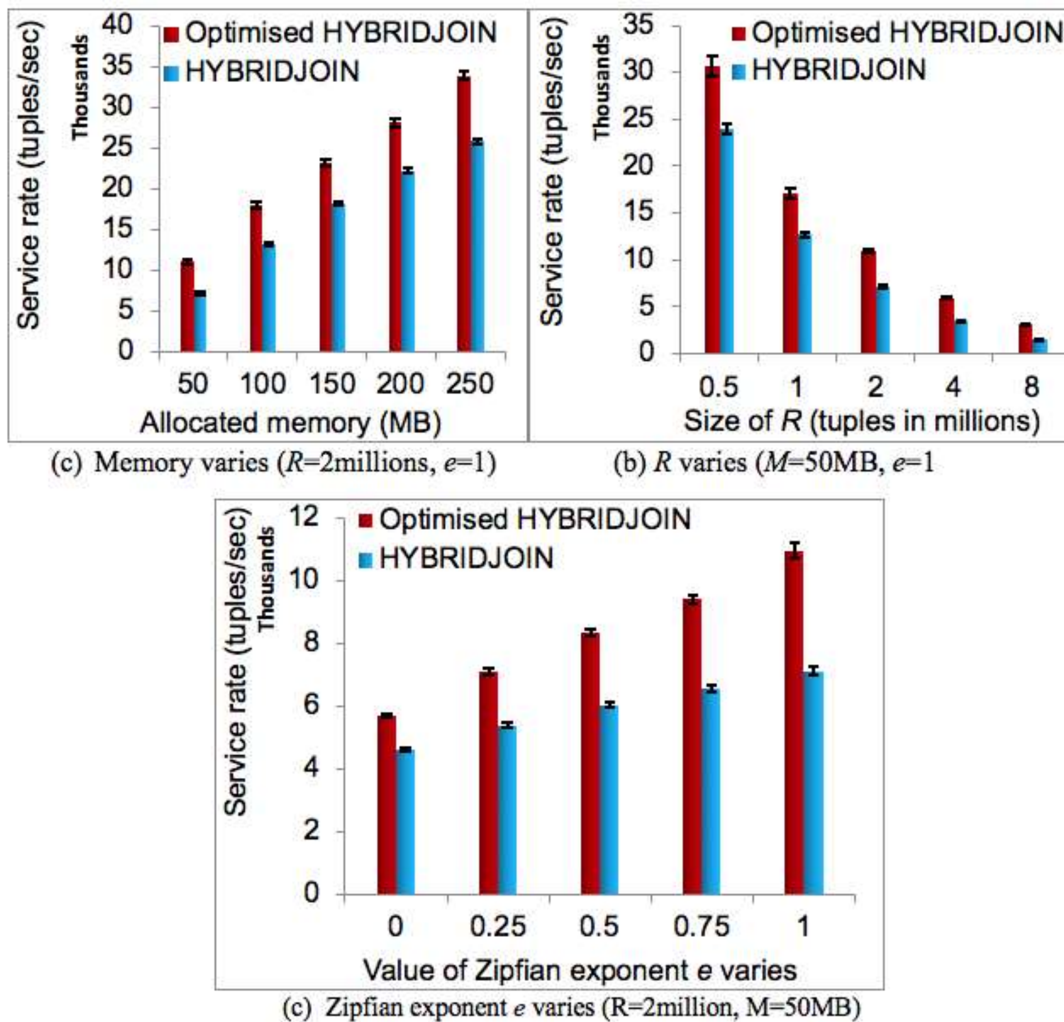
In this section we compared the performance of Optimised HYBRIDJOIN with existing HYBRIDJOIN in terms of service rate by varying the three parameters: the total memory available for the algorithm *M*, the size of *R*, and the value of the parameter skew *e* (also called Zipfian exponent) in the stream data. For the sake of brevity, we restricted the discussion for each parameter to a one-dimensional variation, i.e. we varied one parameter at a time.

In experiment shown in Figure 5(a) we varied memory sizes while the size of *R* was fixed (2 million tuples). From the figure it is clear that for all memory budgets the performance of Optimised HYBRIDJOIN is significantly better than HYBRIDJOIN.

In our second experiment we assumed the total allocated memory for join was fixed while the size of *R* varied exponentially. The results of this experiment are presented in Figure 5(b). From the figure again it can be observed that for the small size of *R* performance of Optimised HYBRIDJOIN is significantly better than HYBRIDJOIN. However, this factor of improvement decreases by increasing the size of *R*. The plausible reason for this behaviour is by increasing the size of *R* the probability of matching the stream tuples against a disk partition decreases while the disk I/O cost remains same because of fixed size of disk partition.

Finally, we evaluated the performance of Optimised HYBRIDJOIN by varying the skew in the stream data. To vary the skew, we varied the value of the Zipfian exponent *e* (Knuth, 1998). In our experiments we allowed it to range 0 to 1. At 0 the input stream was uniform while at 1 the stream had maximum skew. Figure 5(c) presents the results of our experiment. It is clear from the figure that under each value of *e* Optimised HYBRIDJOIN again performs considerably better than HYBRIDJOIN. Also, this improvement increases by increasing the value of *e*. The plausible reason for this behavior is when the value of *e* increases the input stream gets

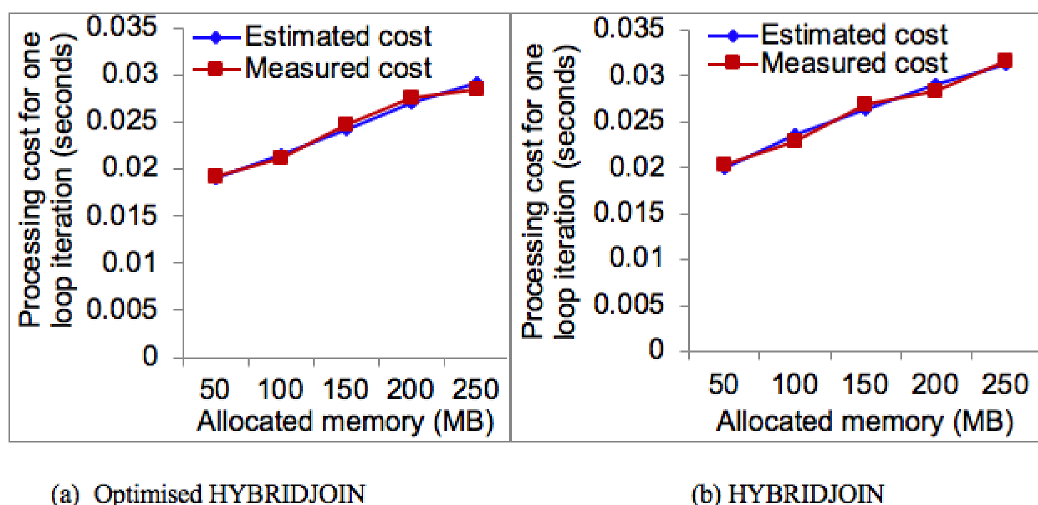
more skewed and consequently the probability of matching the stream tuples against a disk partition increase. In all our experiments we observed that Optimised HYBRIDJOIN outperformed HYBRIDJOIN due to optimising its disk I/O cost which strengthened our argument presented in the paper.



(c) Memory varies (R=2millions, e=1) (b) R varies (M=50MB, e=1

(c) Zipfian exponent e varies (R=2million, M=50MB)

Figure 5: Performance comparison (Optimised HYBRIDJOIN vs. HYBRIDJOIN)



(a) Optimised HYBRIDJOIN (b) HYBRIDJOIN

Figure 6: Cost validation

Cost validation: In this experiment we validated the cost model for the both algorithms by comparing the estimated cost with the measured cost. In the case of the predicted cost, we first determined the cost for one loop iteration using Equation (2). Figure 6 presents the comparisons of both costs for the both algorithms. In the figure it can be observed that in case of the both algorithms the predicted cost closely resembles the measured cost which validates the correctness of our implementations.

6. Conclusions

The disk I/O cost is typically considered a predominant cost among all other join's components cost. The key challenge is therefore to minimize this I/O cost that in turns helps to minimise the delay in processing of the stream data and consequently it improves the service rate. In this paper we presented an optimal approach for accessing the disk-based master data. To test our approach, we extended the existing HYBRIDJOIN algorithm by adding another disk buffer for loading the master data. The new algorithm called Optimised HYBRIDJOIN implements its two phases known as probing phase and loading phase. These two phases execute independently in parallel which is not the case in existing HYBRIDJOIN. By implementing this feature, the new algorithm exploits the allocated memory and CPU resources optimally. We evaluated the performance of the both algorithms and our experiments have shown that the new algorithm significantly outperformed existing HYBRIDJOIN. We also validated our cost models for the both algorithms in order to validate the correctness of our implementations.
In future we have a plan to parallelise our new algorithm by distributing the master data physically and process this through multiple CPUs. The other aim is to extend our algorithm for multi-stage join where the data stream needs to join with multiple tables from the master data.

References

Bornea, M. A., Deligiannakis, A., Kotidis, Y., & Vassalos, V. (2011). Semi-Streamed Index Join for near-real time execution of ETL transformations. In *IEEE 27th International Conference on Data Engineering (ICDE'11)* (pp. 159–170). https://doi.org/10.1109/ICDE.2011.5767906

Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Gupta, A., Haas, L., … Shan, M.-C. (2010). A demonstration of the MaxStream federated stream processing system. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on* (pp. 1093–1096). IEEE.

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 & Distributed Processing* (pp. 1–11). Washington, DC, USA: IEEE Computer Society. https://doi.org/http://dx.doi.org/10.1109/IPDPS.2009.5161064

Chakraborty, A., & Singh, A. (2010). A disk-based, adaptive approach to memory-limited computation of windowed stream joins. In *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part I* (pp. 251–260). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://portal.acm.org/citation.cfm?id=1881867.1881892

Derakhshan, R., Sattar, A., & Stantic, B. (2013). A new operator for efficient stream-relation join processing in data streaming engines. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management* (pp. 793–798).

ACM.
Golfarelli, M., & Rizzi, S. (2009). A survey on temporal data warehousing. *International Journal of Data Warehousing*, *5*.
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* (pp. 28–39). New York, NY, USA: ACM. https://doi.org/http://doi.acm.org/10.1145/1077501.1077509
Knuth, D. E. (1998). *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.
Naeem, M. A., Dobbie, G., & Weber, G. (2011a). HYBRIDJOIN for Near-real-time Data Warehousing. *International Journal of Data Warehousing and Mining (IJDWM)*, *7*(4), 21–42.
Naeem, M. A., Dobbie, G., & Weber, G. (2011b). X-HYBRIDJOIN for near-real-time data warehousing. In *Proceedings of the 28th British national conference on Advances in databases* (pp. 33–47). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=2075914.2075919
Naeem, M. A., Dobbie, G., & Weber, G. (2012). Optimised X-HYBRIDJOIN for near-real-time data warehousing. In *Proceedings of the Twenty-Third Australasian Database Conference (ADC 2012), Melbourne, Australia* (pp. 21–30). Conferences in Research and Practice in Information Technology (CRPIT).
Naeem, M. A., Dobbie, G., Weber, G., & Alam, S. (2010). R-MESHJOIN for Near-real-time Data Warehousing. In *DOLAP'10: Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP*. Toronto, Canada: ACM. https://doi.org/http://dx.doi.org/10.1109/IPDPS.2009.5161064
Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., & Frantzell, N. (2008). Meshing Streaming Updates with Persistent Data in an Active Data Warehouse. *IEEE Trans. on Knowl. and Data Eng.*, *20*(7), 976–991. https://doi.org/http://dx.doi.org/10.1109/TKDE.2008.27
Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., & Frantzell, N. E. (2007). Supporting Streaming Updates in an Active Data Warehouse. In *ICDE 2007: Proceedings of the 23rd International Conference on Data Engineering* (pp. 476–485). Istanbul, Turkey.
Ramakrishnan, R., & Gehrke, J. (2000). *Database management systems*. McGraw-Hill.
Slavin, R. E. (1980). Cooperative learning. *Review of Educational Research*, *50*(2), 315–342.
Thiele, M., Fischer, U., & Lehner, W. (2009). Partition-based workload scheduling in living data warehouse environments. *Information Systems*, *34*(4–5), 382–399.
Tho, M. N., & Tjoa, A. M. (2004). Zero-latency data warehousing for heterogeneous data sources and continuous data streams. In *5th International Conference on Information Integrationand Web-based Applications Services* (pp. 55–64).
Thomsen, C., & Pedersen, T. B. (2005). A survey of open source tools for business intelligence. In *DaWaK* (Vol. 5, pp. 74–84). Springer.
Vassiliadis, P. (2009). A survey of Extract–transform–Load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, *5*(3), 1–27.