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

[1]M. Asif Naeem [2]Gillian Dobbie [2]Gerald Weber

Department of Computer Science, The University of Auckland,
Private Bag 92019, 38 Princes St, Auckland, New Zealand.
Email: [1]mnae006@aucklanduni.ac.nz,[2]{gill,gerald}@cs.auckland.ac.nz

Abstract

Stream-based join algorithms are needed in modern near-real-time data warehouses. A particular class of stream-based join algorithms, with MESHJOIN as a typical example, computes the join between a stream and a disk-based relation. Recently we have presented a new algorithm X-HYBRIDJOIN (Extended Hybrid Join) in that class. X-HYBRIDJOIN achieves better performance compared to earlier algorithms by pinning frequently accessed data from the disk-based relation in main memory. Apart from being held in main memory, X-HYBRIDJOIN treats this frequently accessed data no differently than other data from the disk-based relation. In this paper we investigate whether performance can be improved by treating the frequently accessed data differently. We present a new algorithm called Optimised X-HYBRIDJOIN, which consists of two phases. One phase, called the stream-probing phase, deals with the frequently accessed part of the disk-based relation. The other one is called the disk-probing phase and deals with the other part of the disk-based relation. In experiments we found that the performance of Optimised X-HYBRIDJOIN is significantly better than the performance of X-HYBRIDJOIN. We derive the cost model for our algorithm, which allows us to tune the components of Optimised X-HYBRIDJOIN. We performed an experimental study and we validate the cost model against the experimental results.

1 Introduction

Near-real-time data warehousing plays nowadays a prominent role in supporting overall business decision making. By extending data warehouses from static data repositories to active data repositories, businesses and other organizations can inform their users better and allow them to take effective and timely decisions.

In near-real-time data warehousing the changes occurring at source level are reflected in data warehouses without any delay. Extraction, Transformation, and Loading (ETL) tools are used to access and manipulate transactional data and then load them into the data warehouse. An important phase in the ETL process is a transformation where the source level changes are mapped into the data warehouse format. 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*) there is some information added, namely sales price denoted by *s_price* to calculate the total amount, and the vendor information. In the figure these 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.

In traditional data warehousing the source updates are buffered and the join is performed off-line. On the other hand, in near-real-time data warehousing this operation needs to be performed as soon as the data are received from the data sources. In implementing the online execution of join, one important challenge is the different character 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.

A stream-based join algorithm called X-HYBRIDJOIN (Extended Hybrid Join) (Naeem et al. 2011) was proposed to deal with these challenges. In addition, the algorithm is designed to be particular efficient for Zipfian distributions as they are frequently found in practice. A frequently cited rule of thumb is the 80/20 rule (Anderson 2006). According to this rule 80% of sales in an e-commerce setting is based on 20% of the products and therefore, a small number of pages in master data are frequently used during the join operation. The algorithm used a buffer to load a specific portion of master data into memory. To eliminate the bottleneck in the stream the algorithm divides this buffer into two equal segments. One segment is non-swappable and holds a small number of the frequently accessed page(s) of master data permanently in memory while the other segment is swappable and exchanges its contents on each iteration of the algorithm. The main argument presented in the algorithm is that storing the frequently accessed part of the master data permanently in memory minimises the disk I/O cost and that eventually amortises the fast incoming stream of updates. An alternative approach would be to try to put the whole disk-based relation into memory. In some cases this alternative can be feasible. But 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

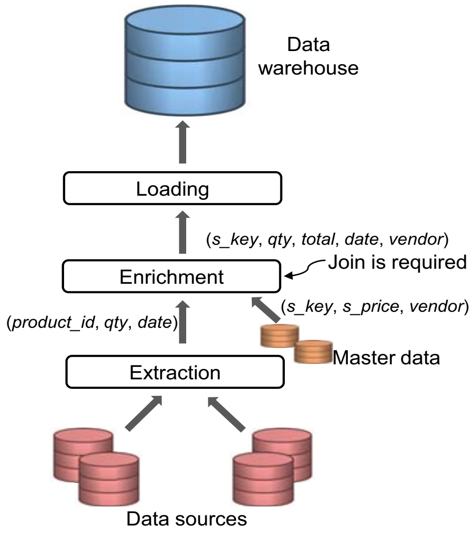


Figure 1: An example of content enrichment

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, in contrast there is no such waste of resources.

In X-HYBRIDJOIN, introducing the non-swappable part in the disk buffer reduces the disk I/O cost. However, there are some unnecessary processing costs that negatively affect the performance of the algorithm. For example in each iteration the algorithm matches all tuples in the non-swappable part of the disk buffer with the hash table. It increases the unnecessary look-up cost for the algorithm. Similarly, the algorithm stores all stream tuples, whether they join with the swappable or non-swappable part of the disk buffer, in memory increasing the cost in terms of loading and unloading the stream tuples into memory. These overheads in terms of extra costs can be removed by improving the architecture of the algorithm.

After considering these observations, we propose an improved version of X-HYBRIDJOIN known as Optimised X-HYBRIDJOIN (Optimised Extended Hybrid Join). In Optimised X-HYBRIDJOIN we divide the algorithm in two phases and both phases can work independently. One phase deals with the swappable part while the other phase deals with the non-swappable part of the disk-based relation using appropriate data structures. In the proposed algorithm, due to choosing an appropriate architecture all unnecessary costs are eliminated and performance is improved significantly. To make our algorithm more efficient we also present the tuning of the algorithm based on a mathematical cost model.

The rest of the paper is structured as follows. The previous work related to the area is presented in Section 2. Section 3 describes our observations for the current algorithm. In Section 4 we present the proposed algorithm with its execution architecture, pseudo-code, cost model and tuning. The experimental study is described in Section 5 and finally Section 6 concludes the paper.

2 Previous work

Considerable work has been done on executing join queries (Chen et al. 2000) (Liu et al. 2004) (Avnur et al. 2000) (Babcock et al. 2003) (Chandrasekaran et al. 2002) (Dobra et al. 2002). Our focus is particularly on stream-based joins. In stream-based joins we further divide our literature review into two categories. In the first category we overview those join operators where all the inputs are in the form of a stream. In the second category we consider those join algorithms in which one input comes in the form of a stream while the other input comes from disk.

First Category: Symmetric Hash Join (SHJ) (Hong et al. 1991) (Wilschut et al. 1991) has exploited the concepts of the traditional hash join algorithm by eliminating the delay for the input stream. SHJ maintains hash tables for both input streams in memory. Each new tuple from one stream is joined with the other stream stored in a hash table and the output for the joined tuple is generated. After generating the output the tuple is stored in its own hash table. The algorithm can generate the output as early as both matching tuples have arrived. However, it needs to store both inputs in memory.

XJoin (Tolga et al. 2000) is an extended form of SHJ that handles memory overflow by flushing the largest single partition on disk. XJoin presents a three stage strategy to switch its execution state between disk and memory. First priority is given to the memory-resident tuples. During times where there are no incoming stream data, the algorithm executes the second, disk-to-memory phase and lastly deals with the tuples stored on disk (disk-to-disk) in the case when the inputs are terminated. In XJoin duplicate tuples are avoided by using a timestamp approach.

The double Pipelined Hash Join (DPHJ) (Ives et al. 1999) is also an extension of symmetric hash join based on two stages. In the first stage, which is similar to SHJ and XJoin, the algorithm joins the tuples which are in memory. In the second stage the algorithm marks the tuples which are not joined in memory and joins them on the disk. In DPHJ duplication of tuples is possible in the second phase when all tuples from both inputs have been read and the final clean-up join is executed. This algorithm is suitable for medium size data and does not perform well for large data. Hash-Merge Join (HMJ) (Mokbel et al. 2004), also one from the series of symmetric joins, is based on push technology and consists of two phases, hashing and merging.

All three approaches above do not consider the metadata about the stream. Therefore, they are unable to recognize data which is no-longer required and by dropping it the overhead can be reduced. In addition, the join approaches above focus on throughput optimisation while ignoring the other optimisation goals such as the characteristics of stream data which are equally important.

MJoin (Viglas et al. 2003), a generalised form of XJoin, extends the symmetric binary join operators to handle multiple inputs. MJoin uses a separate hash table for each input. On the arrival of a tuple from an input, it is stored in the corresponding hash table and is probed into the rest of the hash tables. It is not necessary to probe all hash tables for each arrival, the sequence of probing stops when a probed tuple does not match in a hash table. The methodology for choosing the correct sequence of probing is determined by performing the most selective probes first. The algorithm uses a coordinated flushing technique that involves flushing the same partition on disk for all inputs. All three stages from XJoin are included in MJoin. To identify the duplicate tuples MJoin uses two timestamps for each tuple, arrival time and departure time from memory.

Early Hash Join (Lawrence et al. 2005) is an improved version of XJoin with a different flushing strategy and a simplified technique to determine the duplicate tuples. EHJ uses a biased flushing strategy that supports flushing the partition with large input first.The technique used in EHJ to determine the duplicate tuples is based on cardinality. For one-to-one and one-to-many relationships the algorithm does not use any timestamp while for many-to-many relationships it requires an arrival timestamp only.

In the approaches above, the algorithms expect all inputs in the form of streams, they are not adaptive in the context of near-real-time data warehousing where one input normally comes from disk.

Second Category: In the near-real-time data warehousing context, there is a need for joins between a stream of source updates and a disk-based master data relation. This scenario naturally arises for near-real-time data warehouses, if an incoming stream of user data has to be joined with master data.

The MESHJOIN (Mesh Join) algorithm (Polyzotis et al. 2007) (Polyzotis et al. 2008) has been 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 in memory at the same time. In each iteration the algorithm loads a disk partition into memory and performs the join with all these stream chunks. The algorithm performs tuning for efficient memory distribution among the join components, but we identified in the past some issues around the access to the disk based relation. Also MESHJOIN cannot deal with intermittency of the stream efficiently.

R-MESHJOIN (reduced Mesh Join) (Naeem et al. 2010) is an enhanced form of MESHJOIN in which one issue related to suboptimal distribution of memory among the join components is resolved. However, R-MESHJOIN implements the same strategy as the MESHJOIN algorithm for accessing the disk-based relation.

A partition-based approach (Chakraborty et al. 2009) has been introduced to deal with intermittency in the stream. 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 do 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.

One recent algorithm, HYBRIDJOIN (Hybrid Join) (Naeem et al. 2011) address the issue of accessing the disk-based relation. An effective strategy to access the disk-based relation is introduced in HYBRIDJOIN. Another advantage of HYBRIDJOIN is that it can deal with bursty streams, which is a limitation of both MESHJOIN and R-MESHJOIN. However, if we consider long-tail distributions, we find that the algorithm can be improved further.

The X-HYBRIDJOIN (Naeem et al. 2011) algorithm that we focus on in this paper is an extension of HYBRIDJOIN. This algorithm has been designed particularly to cope with Zipfian distributions. Although this is an adaptive algorithm and performs better than other similar approaches, there are some limitations at the architectural level that needs to be explored further.

The motivation behind Optimised X-HYBRIDJOIN is to refine the existing approach in order to minimize the bottleneck in the stream of updates.

3 Problem definition

In this section we first give an overview of X-HYBRIDJOIN and then identify the limitations that we observed in this algorithm. Our cost models are based on a non-uniform distribution on foreign keys in the stream data. In a real-world data warehousing scenario, uniform distributions are rarely encountered. Instead, frequencies often follow power laws, also known as Zipfian distributions. While power laws are natural surface properties of large data populations, the exponent that governs the power law can vary. Generally, smaller values of exponent give so-called short tails, bigger values of exponent give long tails. Long tails are interesting for scalable, very large data warehouses, since long tails are tipped to become more important in tomorrow's economy, if consumer behavior diversifies (Anderson 2006).

Before going into further detail we first explain the major components of X-HYBRIDJOIN and the role of each component. Figure 2 presents an overview of X-HYBRIDJOIN showing a queue to store stream tuples, a two-part disk buffer, and the disk-based relation R. In actual the algorithm stores stream tuples in the hash table however, for simplicity we assume that these stream tuples are stored in the queue. The join is a hash join, and will be elaborated as in the more detailed description of Optimised X-HYBRIDJOIN.

The queue allows the random deletion of tuples and is currently implemented using a doubly linked-list data structure. This is used for removing tuples that have been matched. The disk buffer is another important component used to load the disk partition into memory. To make efficient use of relation R by minimizing the disk access cost, the disk buffer is divided into two equal parts. One is called the non-swappable part, and stores a small but frequently accessed portion of relation R in memory permanently. The other part of the disk buffer is swappable and for each iteration it loads the disk partition p_i from relation R into memory.

The key idea behind how X-HYBRIDJOIN works is that before the actual execution starts, the algorithm loads the frequently used part of relation R into the non-swappable part of the disk buffer. After the actual execution starts, for each iteration the algorithm reads the oldest tuple from the queue and using this tuple as an index it loads the relevant disk partition into the disk buffer. Then the algorithm matches one-by-one all disk tuples available in both the swappable and non-swappable parts of the disk buffer with the stream tuples in the queue. If the matching is true, the algorithm generates that stream tuple as an output after deleting it from the queue. In the next iteration the algorithm again reads the oldest tuple from the queue, loads the relevant disk partition into the disk buffer and repeats the entire procedure.

Although in X-HYBRIDJOIN, introducing the new component called non-swappable part of disk buffer reduced the disk access cost, it also raised some issues related to processing cost. Firstly, for each iteration, the algorithm looks-up one-by-one all the disk tuples stored in both the swappable and the non-swappable parts of the disk buffer in the hash table. It increases the unnecessary look-up cost for the algorithm particularly when the corresponding stream tuple does not exist in the hash table. Secondly, the algorithm stores all stream tuples, whether they join with the swappable or the non-swappable part of the disk buffer, in memory. As a result, it introduces extra processing costs for algorithms in terms of loading these tuples into the hash table and removing them

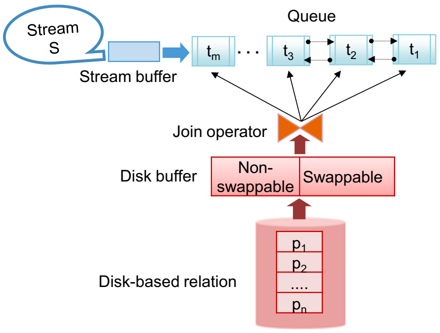


Figure 2: X-HYBRIDJOIN working overview

from the hash table after processing. Contrarily, if we store only those stream tuples in memory that join with the swappable part of the disk buffer, we can accommodate more stream tuples in memory at the same time.

In summary, the problem that we consider in this paper is, how can we eliminate these unnecessary processing costs that occur in X-HYBRIDJOIN by improving its architecture and consequently the way the algorithm works.

4 Proposed solution

In this section, we propose a new algorithm called Optimised X-HYBRIDJOIN (Optimised Extended Hybrid Join) that overcomes the problems that we identified in X-HYBRIDJOIN. Optimised X-HYBRIDJOIN decomposes the algorithm into two hash join phases that can execute separately. One phase uses R as the probe input; the largest part of R will be stored in tertiary memory. This phase is called the disk-probing phase. The other join phase uses the stream as the probe input and it is called the stream-probing phase. This phase deals only with a small part of relation R. For each incoming stream tuple, Optimised X-HYBRIDJOIN first uses the stream-probing phase to find a match for frequent requests quickly, and if no match is found, the stream tuple is forwarded to the disk-probing phase. The details of the proposed algorithm are presented in the following subsections.

4.1 Memory architecture

This section gives a high-level description of Optimised X-HYBRIDJOIN, while a detailed walkthrough of the algorithm can be found in Section 4.2. From the architectural point of view, the key concept in Optimised X-HYBRIDJOIN is to execute both the disk-probing phase and the stream-probing phase independently, using appropriate data structures. The reason for doing this is to eliminate unnecessary costs, as described later in this section. The memory architecture for Optimised X-HYBRIDJOIN is shown in Figure 3. The largest components of Optimised X-HYBRIDJOIN with respect to memory size are two hash tables, one storing stream tuples, denoted by H_S, and the other storing tuples from the disk-based relation, denoted by H_R. The other main components of Optimised X-HYBRIDJOIN are a disk buffer, a queue and a stream buffer. Disk-based relation R and stream S are the external input sources. Similar to X-HYBRIDJOIN R is assumed to be sorted according to the frequency of access. The hash table H_R contains the frequently-accessed part of R, which is stored permanently in memory.

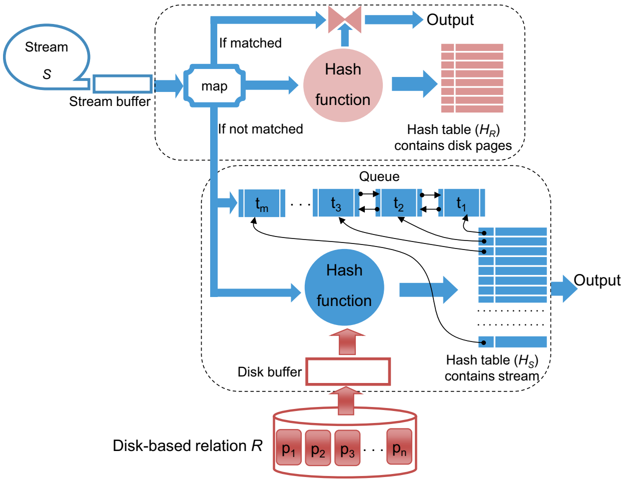

Stream S
Stream buffer
map
If matched
If not matched
Output
Hash function
Hash table (H_R) contains disk pages
Queue
t_m . . . t_3 t_2 t_1
Hash function
Output
Hash table (H_S) contains stream
Disk buffer
Disk-based relation R
p_1 p_2 p_3 . . . p_n

Figure 3: Memory architecture for Optimised X-HYBRIDJOIN

Optimised X-HYBRIDJOIN alternates between the stream-probing and disk-probing phases. The hash table H_S is used to store only that part of the update stream which does not match tuples in H_R. A stream-probing phase ends if H_S is completely filled or if the stream buffer is empty. Then the disk-probing phase becomes active. The length of the disk-probing phase is determined by the fact that only a small number of disk pages of R have to be loaded at one time in order to amortise the costly disk access. In the disk-probing phase of Optimised X-HYBRIDJOIN, the oldest tuple in the queue is used to determine the partition of R that is loaded for a single probe step. This is also the step where Optimised X-HYBRIDJOIN needs an index on table R in order to find the partition in R that matches the oldest stream tuple. After one probe step, a sufficient number of stream tuples are deleted from H_S, so the algorithm switches back to the stream-probing phase. One phase of stream-probing with a subsequent phase of disk-probing constitutes one outer iteration of Optimised X-HYBRIDJOIN. The disk-probing phase could work on its own, without the stream-probing phase. Therefore, the stream-probing phase can be switched-off if it is not required and the memory needed for that phase would be reassigned. The stream-probing phase is used to boost the performance of the algorithm by quickly matching the frequently-used master data. The disk buffer stores the swappable part of R and for each iteration it loads a particular partition of R into the memory. The other component queue is based on a doubly-linked-list, and is used to store the values for the join attribute. Each node in the queue also contains the addresses of its neighbour nodes. The reason for choosing this data structure is to allow random deletion from the queue. The stream buffer is included in the diagram for completeness, but is in reality always a tiny component and will not be considered in the cost model.

There are two key advantages of Optimised X- HYBRIDJOIN over X-HYBRIDJOIN. First, due to the independent processing of each phase the stream tuples can be looked-up directly in H_R without loading them into memory. This not only eliminates an unnecessary look-up cost, but also allows more of the stream to be accommodated in memory. In contrast to this, X-HYBRIDJOIN stores a major part of the stream, related to the non-swappable part, in memory and for each iteration, the algorithm looks-up all the tuples of the non-swappable part in the hash table one-by-one. In the situation when the tuples do not match, the algorithm faces an additional look-up cost. Secondly, since Optimised X- HYBRIDJOIN does not store a large part of the stream in memory, it eliminates the costs of loading and unloading that part of the stream into the hash table, H_S. These additional features in Optimised X-HYBRIDJOIN help in reducing the overall processing cost for the algorithm and that eventually improves the performance.

4.2 Algorithm

After dividing the available memory among the join components, the algorithm starts its execution. The pseudo-code for Optimised X-HYBRIDJOIN is shown in Algorithm 1. The outer loop of the algorithm is an endless loop (line 2). The body of the outer loop has two main phases, the stream-probing phase and the disk-probing phase. Due to the endless loop, these two phases are executed alternately.

Lines 3 to 11 comprise the stream-probing phase. The stream-probing phase has to know the number of empty slots in H_S. This number is kept in variable *hSavailable*. At the start of the algorithm, all the slots in H_S are empty (line 1). The stream-probing phase has an inner loop that continues while stream tuples as well as empty slots in H_S are available (line 3). In the loop, the algorithm reads one input stream tuple t at a time (line 4). The algorithm looks up t in H_R (line 5). In the case of a match, the algorithm generates the join output without storing t in H_S (line 6). In the case where t does not match, the algorithm loads t into H_S, along with enqueuing its key attribute value in the queue (line 8). The counter of empty slots in H_S then has to be decreased (line 9).

Lines 12 to 21 comprise the disk-probing phase. At the start of this phase, the algorithm reads the oldest key attribute value from the queue and loads a partition of R into the disk buffer, using that key attribute value as an index (lines 12 and 13). In an inner loop, the algorithm looks up all tuples r from the disk buffer in hash table H_S one-by-one. In the case of a match, the algorithm generates the join output (line 16). Since H_S is a multi-hash-map, there can be more than one match, the number of matches being f (line 17). The algorithm removes all matching tuples from H_S along with deleting the corresponding nodes from the queue (line 18). This creates empty slots in H_S (line 19). In the next outer iteration the algorithm fills these empty slots if stream input is available.

4.3 Cost calculation

In this section we develop the cost model for our proposed Optimised X-HYBRIDJOIN. The main objective for developing our cost model is to interrelate the key parameters like the algorithm input size w, processing cost c_{loop} for these w tuples, the available memory M and the service rate μ. The other important application for our cost model is in the tuning process where the optimal size is determined for each component of the algorithm. The details about the tuning process are presented in Section 4.4. Normally, the main costs for an algorithm are described in terms of the distribution of memory to the components and processing time. We calculate both of these costs for our proposed Optimised X-HYBRIDJOIN. Equation 1 represents the total memory used by the algorithm except the stream buffer, and Equation 2 describes the processing cost for each iteration of the algorithm. The notations we used in our cost model are specified in Table 1.

Algorithm 1 Optimised X-HYBRIDJOIN

Input: A disk based relation R with an index on join attribute and a stream of updates S.
Output: $R \bowtie S$
Parameters: w (where w=w_S+w_N)tuples of S and k pages of R.
Method:

```
1:  $hSavailable \leftarrow h_S$
2:  while (true) do
3:    while (stream available AND $hSavailable > 0$) do
4:      READ a stream tuple $t$ from the stream buffer
5:      if $t \in H_R$ then
6:        OUTPUT $t \bowtie H_R$
7:      else
8:        ADD the stream tuple $t$ into $H_S$ while also placing its join attribute values into $Q$
9:        $hSavailable \leftarrow hSavailable - 1$
10:     end if
11:   end while
12:   READ the oldest join attribute value from $Q$
13:   READ a partition of $R$ into the disk buffer using the oldest join attribute value from the queue for the index look-up.
14:   for each tuple $r$ in the disk buffer do
15:     if $r \in H_S$ then
16:       OUTPUT $r \bowtie H_S$
17:       $f \leftarrow$ number of matching tuples found in $H_S$
18:       DELETE all matched tuples from $H_S$ along with the corresponding nodes from $Q$
19:       $hSavailable \leftarrow hSavailable + f$
20:     end if
21:   end for
22: end while
```

4.3.1 Memory cost

The optimal size for hash table H_R can be different from the optimal size of the disk buffer. Therefore, we distinguish between k, the number of pages for the disk buffer, and l, the number of pages for H_R. The major portion of the total memory is assigned to the both hash tables while a much smaller portion comparatively is assigned to the disk buffer and the queue. The memory for each component can be calculated as given below.
Memory for disk buffer = $k{\cdot}v_P$
Memory for H_R =$l{\cdot}v_P$
Memory for H_S =$\alpha[M-(k+l)v_P]$
Memory for the queue = $(1-\alpha)[M-(k+l)v_P]$
By aggregating the above, the total memory used by Optimised X-HYBRIDJOIN can be calculated as shown in Equation 1.

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

Currently, the memory for the stream buffer is not included because it is small (0.05 MB is sufficient in all our experiments).

4.3.2 Processing cost

In this section we calculate the processing cost for the algorithm. To make it simple we first calculate the processing cost for individual components and then

Table 1: Notations used in cost estimation of Optimised X-HYBRIDJOIN

Parameter name	**Symbol**
Total allocated memory *(bytes)*	M
Service rate *(processed tuples/sec)*	μ
Number of stream tuples processed in each iteration through H_R	w_N
Number of stream tuples processed in each iteration through H_S	w_S
Stream tuple size *(bytes)*	v_S
Disk page size *(bytes)*	v_P
Size of disk tuple *(bytes)*	v_R
Disk buffer size *(pages)*	k
Disk buffer size *(tuples)*	$d = k\frac{v_P}{v_R}$
Size of H_R *(pages)*	l
Size of H_R *(tuples)*	$h_R = l\frac{v_P}{v_R}$
Size of H_S *(tuples)*	h_S
Disk relation size *(tuples)*	R_t
Memory weight for the hash table	α
Memory weight for the queue	$1-\alpha$
Cost to read k number of disk pages into the disk buffer *(nano secs)*	$c_{I/O}(k{\cdot}v_P)$
Cost to look-up one tuple into the hash table *(nano secs)*	c_H
Cost to generate the output for one tuple *(nano secs)*	c_O
Cost to remove one tuple from the hash table and the queue *(nano secs)*	c_E
Cost to read one stream tuple into the stream buffer *(nano secs)*	c_S
Cost to append one tuple in the hash table and the queue *(nano secs)*	c_A
Total cost for one loop iteration of the algorithm *(secs)*	c_{loop}

sum up all these costs to calculate the total processing cost for one iteration.

$c_{I/O}(l \cdot v_P)$ =Cost to read frequent l number of pages of R into H_R

$c_{I/O}(k \cdot v_P)$ =Cost to read k number of pages (one disk partition) into the disk buffer

$w_N \cdot c_H$ =Cost to look-up w_N tuples in H_R

$d \cdot c_H$ =Cost to look-up the disk buffer tuples in H_S

$w_N \cdot c_O$ =Cost to generate the output for w_N tuples

$w_S \cdot c_O$ =Cost to generate the output for w_S tuples

$w_N \cdot c_S$ =Cost to read the w_N tuples from the stream buffer

$w_S \cdot c_S$ =Cost to read the w_S tuples from the stream buffer

$w_S \cdot c_A$ =Cost to append w_S tuples into H_S and the queue

$w_S \cdot c_E$ =Cost to delete w_S tuples from H_S and the queue

The hash table H_R is filled only once before the actual execution of the algorithm starts; therefore, we exclude it from the iteration cost. By aggregation, the total cost for one loop iteration is:

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

Since in every c_{loop} seconds the algorithm processes w_N and w_S tuples of the stream S, the service rate μ can be calculated using equation 3.

$$\mu = \frac{w_N + w_S}{c_{loop}} \quad (3)$$

4.4 Tuning

Normally the stream-based join algorithms are executed online, where limited memory resources are available. Due to the fixed and small amount of available memory, each component in the join faces a trade-off with respect to memory distribution. Assigning more memory to one component means assigning less memory to some other components. On close observation it can be seen that the components like both hash tables require more memory compared to the other components, such as the disk buffer, the stream buffer and the queue.

The disk buffer and the hash table H_R are the key components for tuning, and the memory assigned to the other components depends on them. The reason for tuning the disk buffer is that the dominant I/O cost is directly connected to the disk buffer.

Tuning is not performed merely using a theoretical approach, rather the optimal tuning settings are approximated using an empirical approach. Finally the experimentally-obtained tuning results are compared with the results obtained using the cost model. Before proceeding further it is first necessary to describe the hardware and software specifications for our experiments.

4.4.1 Experimental arrangement

The details about the experimental setup that we used to implement the prototypes for all comparable algorithms are given below.

Hardware specifications: We accomplished our experiments on *Pentium-IV* machine with *3G* main and *160G* disk memory under *WindowsXP*. We implemented the experiment in *Java* using the *Eclipse IDE 3.3.1.1*. We also used built-in plugins, provided by *Apache*, and *nanoTime()*, provided by *Java API*, to measure the memory and processing time respectively.

Data specifications: We analysed the performance of the algorithms using synthetic data. The relation R is stored on disk using a *MySQL 5.0* database while the bursty stream is generated at run time using our own benchmark algorithm described in (Naeem et al. 2011) with an exponent value equal to 1. As in X-HYBRIDJOIN we also assume that the disk-based relation R is sorted according to the access frequency. To measure the I/O cost more accurately we set the fetch size for *ResultSet* equal to the disk buffer size.

Currently the Optimised X-HYBRIDJOIN supports join for one-to-one and one-to-many relationships. In order to implement the join for one-to-many relationships it needs to store multiple values in the hash table against one key value. However the hash table provided by *Java API* does not support this feature therefore, we used *Multi-Hash-Map*, provided by *Apache Common Collections*, to implement the hash table in our experiments. The detailed specifications of the data set that we used for analysis is shown in Table 2.

Measurement strategy: The performance or service rate of the join is measured by calculating the number of tuples processed in a unit second. In each experiment the algorithm runs for one hour and we start our measurements after 10 minutes and continue it for 30 minutes. For more accuracy we take

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 the queue	12 *bytes*
Benchmark	
Based on	Zipf's law
Characteristics	Bursty and self-similar

three readings for each specification and then take their average as a final result. Where required we also calculate the confidence interval by considering 95% accuracy. Moreover, during the execution of the algorithm it is assumed that no other application is run in parallel.

4.4.2 Tuning using Empirical Approach

This section focuses on the tuning of key components, namely the disk buffer and the hash table H_R using an empirical approach. The performance of the algorithm has been tested for a set of values for both components, rather than for every consecutive value. It has been assumed that the total allocated memory and the size of the disk-based relation are fixed. The sizes for the disk buffer and the hash table H_R are varied in such a way that for each size of the disk buffer the performance is measured against a series of values for the size of H_R. The performance measurements for the grid of values for the sizes of the disk buffer denoted by d and the size of H_R denoted by h_R are shown in Figure 4. The figure shows that, if the performance for each fixed value of d is observed against all values of h_R, in the beginning the performance increases rapidly with an increase in h_R. However, after reaching a particular value of h_R, the performance starts decreasing with further increases in h_R. A plausible reason for this behavior is that initially, increasing h_R increases the probability of matching the stream tuples with H_R rapidly. After attaining the optimal value, further incrementing h_R makes no significant difference to the stream-matching probability, due to the skew factor in stream distribution. On the other hand, the associated reduction in memory size for the hash table H_S means that the performance begins to decrease. Similarly when the performance is analysed for each fixed value of h_R against all the values of d, initially the performance increases, since the costly disk access is amortised for a larger number of stream tuples. After attaining a maximum, the performance decreases because of the increase in I/O cost for loading more of R at one time in a non-selective way.

The figure shows that the optimal memory settings for both the disk buffer and the hash table H_R can be determined by considering the intersection of the values of both components at which the algorithm individually performs at a maximum.

4.4.3 Tuning based on cost model

To validate our cost model against measurements, we tune our algorithm based on this cost model as presented in Section 4.3. According to Equation 3 the service rate depends on the values of w_S, w_N and the cost c_{loop}. Therefore, to determine the settings at which the algorithm performs optimally it is first necessary to calculate the sizes of w_N and w_S.

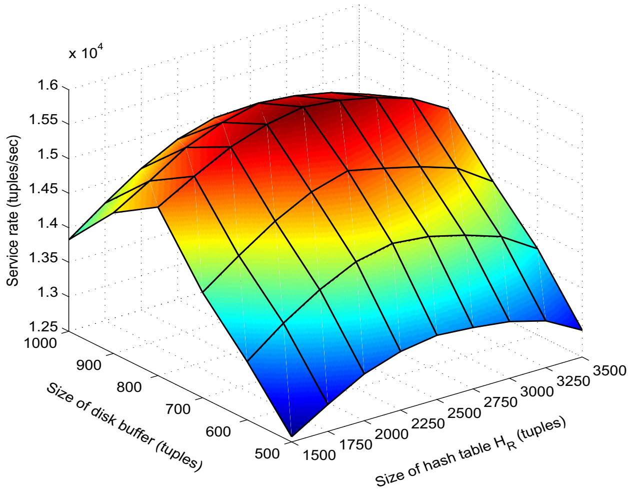


Figure 4: Tuning of Optimised X-HYBRIDJOIN using measurement approach

Mathematical model to calculate w_N: The main components that directly affect w_N are the total size of R (denoted by R_t) on the disk and the size of the hash table H_R (denoted by h_R) that contains the frequently-used part of R in the memory. If the stream of updates S is formulated using Zipfs law with the exponent value being equal to 1, then the matching probability p_N for stream S with H_R can be determined using Equation 4.

$$p_N = \frac{\sum_{x=1}^{h_R} \frac{1}{x}}{\sum_{x=1}^{R_t} \frac{1}{x}} = \frac{\ln(h_R)}{\ln(R_t)} \tag{4}$$

Now using Equation 4 the constant factors of change can be determined in p_N by changing the values of h_R and R_t individually. This assumes that p_N decreases by a constant factor ϕ_N if the value of R_t is doubled, and increases by a constant factor ψ_N if the value of h_R is doubled. Knowing these constant factors the value of w_N can be calculated. Consider a hypothesis

$$p_N = R^y h_R^z \tag{5}$$

where y and z are unknown constants whose values need to be determined.

By doubling R_t, the matching probability p_N decreases by a constant factor ϕ_N, Equation 5 becomes:

$$\phi_N p_N = (2R)^y h_R^z$$

Dividing the above equation by Equation 5 we get $2^y = \phi_N$ and therefore, $y = \log_2(\phi_N)$. Similarly by doubling h_R the matching probability p_N increases by a constant factor ψ_N therefore, Equation 5 can be written as:

$$\psi_N p_N = R^y (2h_R)^z$$

By dividing the above equation by Equation 5 we get $2^z = \psi_N$ and therefore, $z = \log_2(\psi_N)$. After putting the values of constants y and z in Equation 5 we get:

$$p_N = R^{\log_2(\phi_N)} h_R^{\log_2(\psi_N)}$$

If S is the total number of stream tuples that are processed (through both the stream-probing and disk-probing phases) in N iterations, then w_N can be calculated using Equation 6

$$w_N = \frac{(R^{\log_2(\phi_N)} h_R^{\log_2(\psi_N)})S}{N} \qquad (6)$$

Mathematical model to calculate w_S: The second phase of the Optimised X-HYBRIDJOIN algorithm, also called the disk-probing phase, deals with the rest of the disk-based master data R' (where $R' = R_t - h_R$), which occurs less frequently in the stream input as compared to that part which exists permanently in memory. The algorithm reads R' in partitions while the size of each partition is equal to the size of the disk buffer d. As mentioned earlier, the daily market transactions typically formulate the Zipfian distribution, which means that matching probability for every next partition in R' is less than the previous one. Therefore, the matching probability for each partition is calculated by taking the summation over the discrete Zipfian distribution separately and then aggregating all of them as shown below.

$$\sum_{x=h_R+1}^{h_R+d} \frac{1}{x} + \sum_{x=h_R+d+1}^{h_R+2d} \frac{1}{x} + \sum_{x=h_R+2d+1}^{h_R+3d} \frac{1}{x} + \cdots + \sum_{x=h_R+(n-1)d+1}^{h_R+nd} \frac{1}{x}$$

We simplify this to:

$$\sum_{x=h_R+1}^{h_R+nd} \frac{1}{x} \Rightarrow \sum_{x=h_R+1}^{R_t} \frac{1}{x}$$

From this the average matching probability $\overline{p}_S$ can be obtained in the disk probe phase, which is needed for calculating w_S. Let n be the total number of partitions in R', then the average matching probability $\overline{p}_S$ can be determined by dividing the above summation by n. In the denominator, a similar normalization term to that used in Equation 4 is used.

$$\overline{p}_S = \frac{\sum_{x=h_R+1}^{R_t} \frac{1}{x}}{n \sum_{x=1}^{R_t} \frac{1}{x}} = \frac{\ln(R_t) - \ln(h_R + 1)}{n(\ln(R_t) + \gamma)} \qquad (7)$$

To determine the effects of d, h_R and R_t on $\overline{p}_S$ the same number of steps is required as in the case of w_N. If d is doubled then n will be halved in Equation 7 and therefore, the value of $\overline{p}_S$ increases with a constant factor of θ_S. Similarly, if h_R and R_t are doubled one-by-one in Equation 7, the value of $\overline{p}_S$ decreases with a constant factor of ψ_S and ϕ_S respectively. A similar hypothesis is considered here as in Equation 5.

$$\overline{p}_S = d^x h_R^y R_t^z \qquad (8)$$

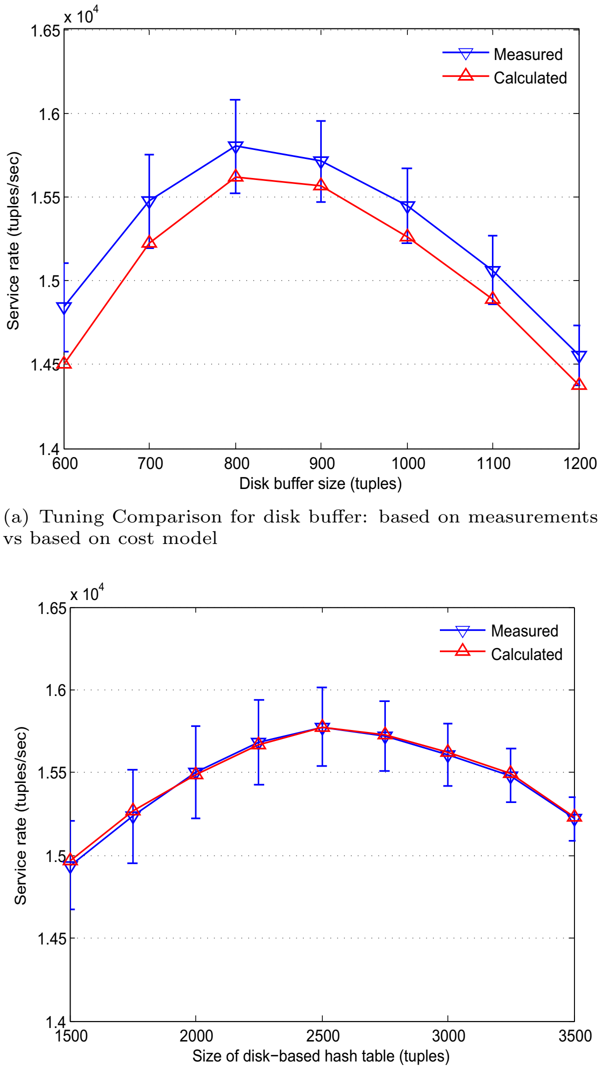
The values for the constants x, y and z in this case will be $x = \log_2(\theta_S)$, $y = \log_2(\psi_S)$ and $z = \log_2(\phi_S)$ respectively. Therefore by replacing the parameters with constants, Equation 8 will become.

$$\overline{p}_S = d^{\log_2(\theta_S)} h_R^{\log_2(\psi_S)} R_t^{\log_2(\phi_S)}$$

If h_S are the number of stream tuples stored in the hash table then the average value for w_S can be calculated using Equation 9.

$$w_S(\text{average}) = d^{\log_2(\theta_S)} h_R^{\log_2(\psi_S)} R_t^{\log_2(\phi_S)} h_S \qquad (9)$$

Once the values of w_N and w_S have been determined, the algorithm can be tuned using Equation 3.



(a) Tuning Comparison for disk buffer: based on measurements vs based on cost model

(b) Tuning Comparison for hash table H_R: based on measurements vs based on cost model

Figure 5: Tuning comparisons for Optimised X-HYBRIDJOIN using both empirical and mathematical approaches

4.4.4 Comparisons of both Tuning Approaches

In this section to validate our cost model, we compare the tuning results obtained through measurements with the tuning results that we calculated using the cost model.

Disk buffer: In this experiment we perform tuning of the disk buffer using both the measurement and the mathematical approaches. The tuning results of each approach are shown in Figure 5(a). From the figure it can be observed that the results in both cases are very similar with a deviation of only 0.38%.

Hash table H_R: We also made the tuning comparisons for hash table H_R using both approaches. The experimental results in this case are shown in Figure 5(b). From the figure, the results in both cases are again closely related with a deviation of only 0.33%. This proves the accuracy of our cost model.

5 Experimental study

In this section we present a series of experimental results to support the proposed join algorithm. We conducted our experiments in two dimensions. In Section 5.1 we compare the performance of Optimised X-HYBRIDJOIN with algorithms that are directly related to it. In Section 5.2 we compare the costs predicted by the cost model for the algorithm with the measured costs.

5.1 Performance evaluation

In the near-real-time data warehousing context the total allocated memory and the size of disk-based relation are the common parameters that can vary and directly affect the performance of the algorithm. Therefore, in our experiments we compare all algorithms by varying both parameters one-by-one.

Performance comparisons when the size of *R* varies: In this experiment we compare the performance of Optimised X-HYBRIDJOIN with other join algorithms. In our experiments we assume that the size of disk-based relation R varies exponentially while the total allocated memory is fixed for all values of R. The performance results are shown in Figure 6(a). From the figure it is clear that for all settings of R the performance in the case of Optimised X-HYBRIDJOIN is significantly better than that of the other algorithms.

Performance comparisons for different memory budgets: In our second experiment we test the performance of all algorithms using different memory budgets by keeping the size of R fixed (*2 million tuples*). Figure 6(b) presents the comparisons of all approaches. From the figure, for all memory budgets, Optimised X-HYBRIDJOIN again performs significantly better than all the other approaches.

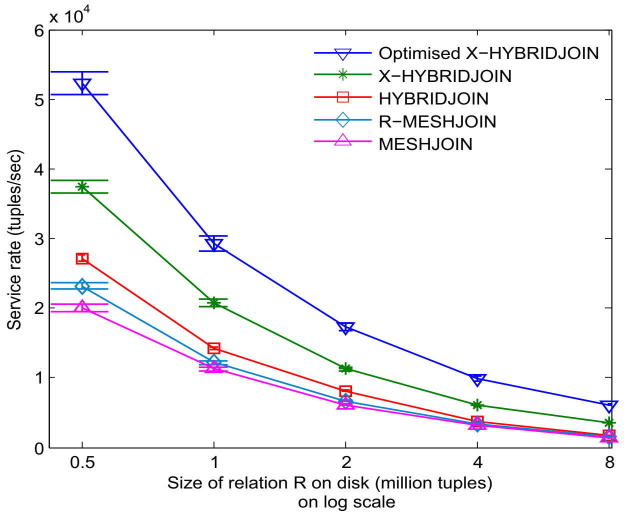
In both scenarios the reason for improvement in performance is the use of an efficient architecture in Optimised X-HYBRIDJOIN. On the contrary, in X-HYBRIDJOIN the data structures used for some components are ineffective causing some unnecessary costs in processing the stream tuples and eventually it effects the performance of the algorithm negatively.

5.2 Cost validation

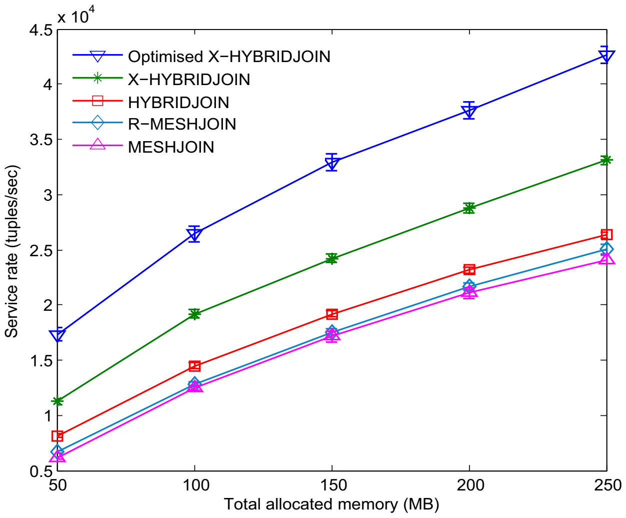
In the second part of our experiments we validate the cost model for the algorithm by comparing the predicted cost with the measured cost. Figure 7 presents the comparisons of cost model predictions and measurements for different memory settings. It can be observed from the figure that for each memory setting the predicted cost is close to the measured cost. This proves the accuracy of the cost model.

6 Conclusions and future work

In this paper we present a significant optimisation for a recently introduced stream-based join called X-HYBRIDJOIN (Extended Hybrid Join). This algorithm is designed to efficiently process non-uniformly distributed data as found in real-world applications. In our investigation we discover that the algorithm has some architectural limitations affecting its performance. Data structures used for some components such as the non-swappable part of the disk buffer, are not optimal, causing additional look-up cost. In addition, the algorithm stores a major part of the stream in memory that matches with the non-swappable part of the disk buffer which is unnecessary and generates extra costs for loading and unloading stream tuples into memory. On the basis of these observations we propose an optimised version of the existing X-HYBRIDJOIN called Optimised X-HYBRIDJOIN (Optimised Extended Hybrid



(a) Size of relation on disk varies



(b) Size of allocated memory varies

Figure 6: Performance comparisons of Optimised X-HYBRIDJOIN with related join algorithms

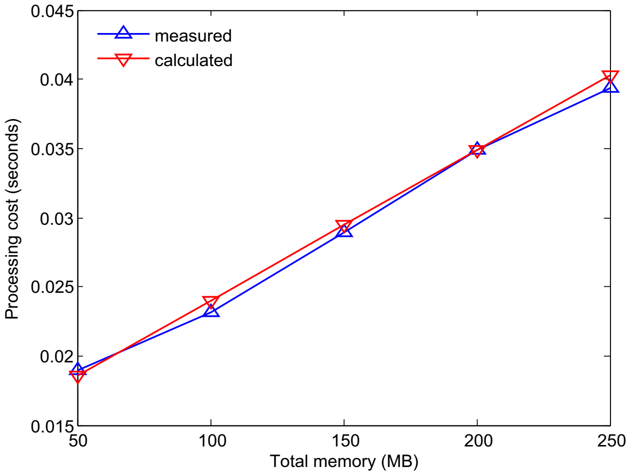


Figure 7: Costs validation

Join). In the proposed algorithm, processing of tuples that match the swappable and non-swappable parts of the master data are executed independently using efficient data structures. The stream that matches with the non-swappable part does not need to be stored in memory. This has two advantages: (a) It eliminates the additional costs required for loading and unloading the stream tuples into memory. (b) More stream tuples that are related to the swappable part can be accommodated in memory. We calculate the mathematical costs for our algorithm and tune the algorithm based on both measurement and cost model. To compare the performance with related algorithms we implemented prototypes of all approaches. Our experiments show that Optimised X-HYBRIDJOIN performs significantly better than the related approaches. We also provide the source code for our implementations.

In the future we plan to generalise our algorithm for other kinds of distributions. This will be particularly useful for markets that do not follow the 80/20 Rule. Additionally, the generalisation of the algorithm will remove the need for the disk-based relation to be sorted.

Source URL: The source of our implementations can be downloaded from the given URL.
https://www.cs.auckland.ac.nz/research/groups/serg/source/

References

Naeem, M. A., Dobbie, G. & Weber, G. (2011), 'HYBRIDJOIN for Near-real-time Data Warehousing', *International Journal of Data Warehousing and Mining (IJDWM)*, IGI Global.

Naeem, M. A., Dobbie, G. & Weber, G. (2011), X-HYBRIDJOIN for Near-real-time Data Warehousing, *in* 'Proceedings of 28th British National Conference on Databases (BNCOD '11)', Springer, Manchester, UK, pp. 33–47.

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

Chen, J., DeWitt, D. J., Tian, F. & Wang, Y. (2000), *in* 'SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data', ACM, New York, NY, USA, pp. 379–390.

Arasu, A., Babcock, B., Babu, S., McAlister, J. & Widom, J. (2004), 'Characterizing memory requirements for queries over continuous data streams', ACM Trans. Database Syst., **29**(1), 162–194.

Avnur, R. & Hellerstein, J. M. (2000), 'Eddies: continuously adaptive query processing', SIGMOD Rec., ACM, **29**(2), 261–272.

Babcock, B., Datar, M., Motwani, R. & O'Callaghan, L. (2003), Maintaining variance and k-medians over data stream windows, *in* 'PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems', ACM, New York, NY, USA, pp. 234–243.

Chandrasekaran, S. & Franklin, M. J. (2002), Streaming queries over streaming data, *in* 'VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases', VLDB Endowment, Hong Kong, China, pp. 203–214.

Dobra, A., Garofalakis, M., Gehrke, J. & Rastogi, R. (2002), Processing complex aggregate queries over data streams, *in* 'SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data', ACM, New York, NY, USA, pp. 61–72.

Wilschut, A. N. & Apers, P. M. G. (1991), Dataflow query execution in a parallel main-memory environment, *in* 'PDIS '91: Proceedings of the first International Conference on Parallel and Distributed Information Systems', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 68–77.

Hong, W. & Stonebraker, M. (1991), Optimization of parallel query execution plans in XPRS, *in* 'PDIS '91: Proceedings of the first International Conference on Parallel and Distributed Information Systems', IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 218–225.

Tolga U. & Michael J. F. (2000), 'XJoin: A reactively-scheduled pipelined join operator', IEEE Data Engineering Bulletin, **23**, 27–33.

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

Mokbel, M. F., Lu, M. & Aref, W. G. (2004), Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results, *in* 'ICDE '04: Proceedings of the 20th International Conference on Data Engineering', IEEE Computer Society Press, Washington, DC, USA, pp. 251.

Viglas, S. D., Naughton, J. F. & Burger, J. (2003), Maximizing the output rate of multi-way join queries over streaming information sources, *in* 'VLDB '2003: Proceedings of the 29th International Conference on Very Large Data Bases', VLDB Endowment, Berlin, Germany, pp. 285–296.

Lawrence, R. (2005), Early Hash Join: A configurable algorithm for the efficient and early production of join results, *in* 'VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases', VLDB Endowment, Trondheim, Norway, pp. 841–852.

Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A. & Frantzell, N.E. (2008), 'Meshing Streaming Updates with Persistent Data in an Active Data Warehouse', IEEE Trans. on Knowl. and Data Eng., **20**(7), 976–991.

Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A. & Frantzell, N.E. (2007), Supporting Streaming Updates in an Active Data Warehouse, *in* 'ICDE 2007: Proceedings of the 23rd International Conference on Data Engineering', IEEE Computer Society, Istanbul, Turkey, pp. 476-485.

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', IEEE Computer Society, Washington, DC, USA, pp. 1–11.

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', ACM, Washington, Toronto, Canada, pp. 53–60.