Meshing Streaming Updates with Persistent Data in an Active Data Warehouse

Neoklis Polyzotis, *Member*, *IEEE*, Spiros Skiadopoulos, Panos Vassiliadis, *Member*, *IEEE*, Alkis Simitsis, and Nils-Erik Frantzell

Abstract—Active Data Warehousing has emerged as an alternative to conventional warehousing practices in order to meet the high demand of applications for up-to-date information. In a nutshell, an active warehouse is refreshed online and thus achieves a higher consistency between the stored information and the latest data updates. The need for online warehouse refreshment introduces several challenges in the implementation of data warehouse transformations, with respect to their execution time and their overhead to the warehouse processes. In this paper, we focus on a frequently encountered operation in this context, namely, the join of a fast stream S of source updates with a disk-based relation R, under the constraint of limited memory. This operation lies at the core of several common transformations such as surrogate key assignment, duplicate detection, or identification of newly inserted tuples. We propose a specialized join algorithm, termed mesh join (MESHJOIN), which compensates for the difference in the access cost of the two join inputs by 1) relying entirely on fast sequential scans of R and 2) sharing the I/O cost of accessing R across multiple tuples of S. We detail the MESHJOIN algorithm and develop a systematic cost model that enables the tuning of MESHJOIN for two objectives: maximizing throughput under a specific memory budget or minimizing memory consumption for a specific throughput. We present an experimental study that validates the performance of MESHJOIN on synthetic and real-life data. Our results verify the scalability of MESHJOIN to fast streams and large relations and demonstrate its numerous advantages over existing join algorithms.

Index Terms—Active data warehouse, join, MESHJOIN, streams, relations.

1 Introduction

Data warehouses are typically refreshed in a batch (or offline) fashion: The updates from data sources are buffered during working hours and then loaded through the Extraction-Transformation-Loading (ETL) process when the warehouse is quiescent (e.g., overnight). This clean separation between querying and updating is a fundamental assumption of conventional data warehousing applications and clearly simplifies several aspects of the implementation. The downside, of course, is that the warehouse is not continuously up to date with respect to the latest updates, which in turn implies that queries may return answers that are essentially stale.

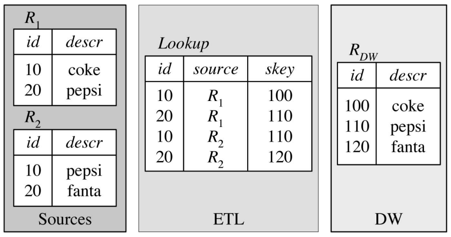
To address this issue, recent works have introduced the concept of *active* (or real-time) *data warehouses* [1], [2], [3], [4]. In this scenario, all updates to the production systems are propagated immediately to the warehouse and incorporated in an online fashion. This paradigm shift raises several challenges in implementing the ETL process, since it implies that transformations need to be performed continuously as update tuples are streamed in the warehouse.

We illustrate this point with the common transformation of surrogate key generation, where the source-dependent key of an update tuple is replaced with a uniform warehouse key. This operation is typically implemented by joining the source updates with a look-up table that stores the correspondence between the two sets of keys. Fig. 1 shows an example, where the keys of two sources (column id in relations R_1 and R_2) are replaced with a warehouse-global key (column $skey$ in the final relation). In a conventional warehouse, the tuples of R_1 and R_2 would be buffered, and the join would be performed with a blocking algorithm in order to reduce the total execution time for the ETL process. An active warehouse, on the other hand, needs to perform this join as the tuples of R_1 and R_2 are propagated from the operational sources. A major challenge, of course, is that the inputs of the join have different access costs and properties: the tuples of R_1 and R_2 arrive at a fast rate and must be processed in a timely fashion, while look-up tuples are retrieved from the disk and are thus more costly to process.

Consider also the case of the identification of newly inserted tuples. Frequently, the population of either smaller dimension tables or bigger parts of the data warehouse is not realized in an incremental fashion. Instead, each time, complete snapshots of the source records are sent to the data staging area, where they are used in order to detect the changes from the previous load. This operation is preferred in various cases, e.g., when the source system should not be

- N. Polyzotis is with the Computer Science Department, University of California at Santa Cruz, MS: SOE3, 1156 High Street, Santa Cruz, CA 95064. E-mail: alkis@soe.ucsc.edu.
- S. Skiadopoulos is with the Department of Computer Science and Technology, University of Peloponnese, Karaiskaki Str., 22100, Tripoli, Hellas. E-mail: spiros@uop.gr.
- P. Vassiliadis is with the Department of Computer Science, University of Ioannina, Ioannina, 45110, Hellas. E-mail: pvassil@cs.uoi.gr.
- A. Simitsis is with Advanced Data Services, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120. E-mail: asimits@us.ibm.com.
- N.-E. Frantzell is with Microsoft Corporation, One Microsoft Way, Redmond, WA 98052. E-mail: n_frantzell@yahoo.com.

Manuscript received 9 Mar. 2007; revised 19 Oct. 2007; accepted 19 Dec. 2007; published online 18 Jan. 2008.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2007-05-0206.
Digital Object Identifier no. 10.1109/TKDE.2008.27.


Sources

R_1

id	descr
10	coke
20	pepsi

R_2

id	descr
10	pepsi
20	fanta

ETL

Lookup

id	source	skey
10	R_1	100
20	R_1	110
10	R_2	110
20	R_2	120

DW

R_{DW}

id	descr
100	coke
110	pepsi
120	fanta

Fig. 1. Surrogate key generation.

overloaded with the extraction task, in the presence of legacy source systems, due to short time windows available for the ETL process, and so forth. In such cases, the difference between the previous snapshot of the data, *R*, and the updated one, *S*, produces either the newly inserted records (*S*-*R*) or the deleted records (*R*-*S*). (In both cases, the updates are included in the result.) In terms of implementation, this operation is a join. For instance, assume the identification of the newly inserted records. For each record of S, a record in R should be probed, and the tuples of S that do not join with R populate the output of the join. In our setting, in the case of active ETL, we have a stream S of tuples probing the existing snapshot, a relation R through a join operator.

The previous examples are the characteristic of several common transformations that take place in an active ETL process such as duplicate detection or difference operation. Essentially, we can identify $S \bowtie_{\mathcal{C}} R$ as a core operation, where S is the relation of source updates, R is a large disk-resident warehouse relation, and the join condition $\mathcal{C}$ depends on the semantics of the transformation. An active warehouse requires the evaluation of this expression online, i.e., as the tuples of S are streamed from the operational sources, in order to ensure that the updates are propagated in a timely fashion. The major challenge, of course, is handling the potentially fast arrival rate of S tuples relative to the slow I/O access of R. Moreover, the join algorithm must operate under limited memory since the enclosing transformation is chained to other transformations that are also executed concurrently (and in the same pipelined fashion).

A natural question is whether we can adapt existing join algorithms to this setting. Consider, for instance, the Indexed Nested Loops (INL) algorithm, where S is accessed one tuple a time (outer input), and R is accessed with a clustered index on the join attribute (inner input). This setup satisfies our requirements, as S does not need to be buffered, and the output of the join is generated in a pipelined fashion. Still, the solution is not particularly attractive since: 1) it may require the (potentially expensive) maintenance of an additional index on R, and most importantly, 2) probing the index with update tuples incurs expensive random I/Os. The latter affects the ability of the algorithm to keep up with the fast arrival rate of source updates and thus limits severely the efficacy of INL as a solution for active data warehousing. We note that similar observations can be made for blocking join algorithms such as sort-merge and hash join. While it is possible to adapt them to this setting, it would require a considerable amount of disk buffering for S tuples, which in turn would slow down the join operation.

Motivated by these observations, we introduce a specialized join algorithm, termed MESHJOIN, that joins a fast update stream S with a large disk resident relation R under the assumption of limited memory. As we stressed earlier, this is a core problem for active ETL transformations, and its solution is thus an important step toward realizing the vision of active data warehouses. MESHJOIN applies to a broad range of practical configurations: it makes no assumption of any order in either the stream or the relation; no indexes are necessarily present; the algorithm uses limited memory to allow multiple operations to operate simultaneously; the join condition is arbitrary (equality, similarity, range, etc.); the join relationship is general (i.e., many-to-many, one-to-many, or many-to-one); and the result is exact. More concretely, our technical contributions can be summarized as follows:

- **MESHJOIN algorithm.** We introduce the MESHJOIN algorithm for joining a fast stream S of source updates with a large warehouse relation R. Our proposed algorithm relies on two basic techniques in order to increase the efficiency of the necessary disk accesses: 1) it accesses R solely through fast sequential scans, and 2) it amortizes the cost of I/O operations over a large number of stream tuples. As we show in this article, this enables MESHJOIN to scale to very high stream rates while maintaining a controllable memory overhead.
- **MESHJOIN performance model.** We develop an analytic model that correlates the performance of MESHJOIN to two key factors, namely, the arrival rate of update tuples and the memory that is available to the operator. In turn, this provides the foundation for tuning the operating parameters of MESHJOIN for two commonly encountered objectives: maximizing processing speed for a fixed amount of memory and minimizing memory consumption for a fixed speed of processing.
- **Approximate join processing.** We examine the use of tuple shedding in order to cope with an update arrival rate that exceeds the service rate of MESHJOIN under the allotted memory. We consider several strategies and the scenarios for which they are suitable, and we examine in more detail the family of strategies that attempt to minimize the absolute number of missed results. In this context, we introduce the TOPW online strategy, and we analyze the optimal offline strategy that has a priori knowledge of the complete update stream.
- **Experimental study of MESHJOIN.** We verify the effectiveness of our techniques with an extensive experimental study on synthetic and real-life data sets of varying characteristics. Our results demonstrate the effectiveness of MESHJOIN and its advantages over existing approaches.

The remainder of the paper is structured as follows: In Section 2, we define the problem more precisely and discuss the requirements for an effective solution. Section 3 provides a detailed definition of the proposed algorithm, including its analytical cost model and its tuning for different objectives. Section 4 examines the problem of approximate join processing and discusses different shedding strategies. We present our experimental study in Section 5 and cover related work in Section 6. We conclude the paper in Section 7.

2 Preliminaries and Problem Definition

We consider a data warehouse and, in particular, the transformations that occur during the ETL process. Several of these transformations (e.g., surrogate key assignment, duplicate detection, or identification of newly inserted tuples) can be mapped to the operation $S \bowtie_{\mathcal{C}} R$, where S is the relation of source updates, R is a large relation stored in the data staging area, and $\mathcal{C}$ depends on the transformation. To simplify our presentation, we henceforth assume that $\mathcal{C}$ is an equality condition over specific attributes of S and R and simply write $S \bowtie R$ to denote the join. As we discuss later, our techniques are readily extensible to arbitrary join conditions.

Following common practice, we assume that R remains *fixed* during the transformation, or alternatively that it is updated only when the transformation has completed. Clearly, there may be cases where R is updated frequently, and this raises the interesting problem of evaluating $S \bowtie R$ in the presence of a concurrent stream of updates. We focus our work on the case where R is fixed since we consider it a first step toward the development of a more general solution. As we see later, this variant of the problem already poses several interesting and challenging technical issues. We make no assumptions about the physical characteristics of R, e.g., the existence of indices or its clustering properties, except that it is too large to fit in the main memory. Thus, the solution that we develop is applicable in a wide range of settings. Of course, it may be possible to design more effective join operators that take into account the particular physical characteristics of R. We consider this to be an interesting venue for future work.

Since our focus is active warehousing, we assume that the warehouse receives S from the operational data sources in an online fashion. Thus, we henceforth model S as a streaming input and use λ to denote the (potentially variable) arrival rate of update tuples. Following common practice, we are interested in cases where the contents of the stream are not affected by changes in other streams of updates coming from other sources; in other words, the records propagated to the mesh-join module are conflict-free, and any possible transactional conflicts are resolved by the appropriate synchronization policies outside the join module [5].

Given our goal of real-time updates, we wish to compute the result of $S \bowtie R$ in a streaming fashion as well, i.e., without buffering S first. (Buffering would correspond to the conventional batch approach.) We assume a restricted amount of available memory M_{max} that can be used for the processing logic of the join. Combined with the (potentially) high arrival rate of S, it becomes obvious that the join algorithm can perform limited buffering of stream tuples in main memory and thus has stringent time constraints for examining each stream tuple and computing its join results. (A similar observation can be made for buffering S tuples on the disk, given the relatively high cost of disk I/O.) We also assume that the available memory is a small fraction of the relation size, and hence, the join algorithm has limited resources for buffering data from R as well.

We consider two metrics of interest for a specific join algorithm: the service rate μ and the consumed memory M. The service rate μ is simply defined as the highest stream arrival rate that the algorithm can handle and is equivalent to the throughput in terms of processed tuples per second. The amount of memory M, on the other hand, relates the performance of the algorithm to the resources that it requires. (We assume that $M \leq M_{max}$.) Typically, we are interested in optimizing one of the two metrics given a fixed value for the other. Hence, we may wish to minimize the required memory for achieving a specific service rate or to maximize the service rate for a specific memory allocation.

Summarizing, the problem that we tackle in this paper involves 1) the introduction of an algorithm that evaluates the join of a fixed disk-based warehouse relation with a stream of source updates without other assumptions for the stream or the relation, and 2) the characterization of the algorithm's performance in terms of the service rate and the required memory resources.

3 Mesh Join

In this section, we introduce the MESHJOIN algorithm for joining a stream S of updates with a large disk-resident relation R. We describe the mechanics of the algorithm, develop a cost model for its operation, and finally discuss how the algorithm can be tuned for two metrics of interest, namely, the arrival rate of updates and the required memory.

3.1 Algorithm Definition

Before describing the MESHJOIN algorithm in detail, we illustrate its key idea using a simplified example. Observe the example in Fig. 1, where a table *Lookup* is employed to assign surrogate keys to incoming factual tuples. Assume a table *Sales* recording sales and a stream of newly inserted source data, which are joined to the tuples of *Lookup* in order to trade their production key (id) with a data warehouse globally unique key ($skey$) before being loaded to relation $Sales$. In the rest, we will refer to the relation *Lookup* as R and to the stream of newly inserted source sales as S. Assume that R contains two pages (p_1 and p_2) and that the join algorithm has enough memory to store a window of the two most recent tuples of the stream. For this example, we will assume that the join processing can keep up with the arrival of new tuples. The operation of the algorithm at different time instants is shown in Fig. 2 and can be described as follows:

- At time $t = 0$, the algorithm reads in the first stream tuple s_1 and the first page p_1 and joins them in memory.
- At time $t = 1$, the algorithm brings in memory the second stream tuple s_2 and the second page p_2. At this point, page p_2 is joined with two stream tuples.

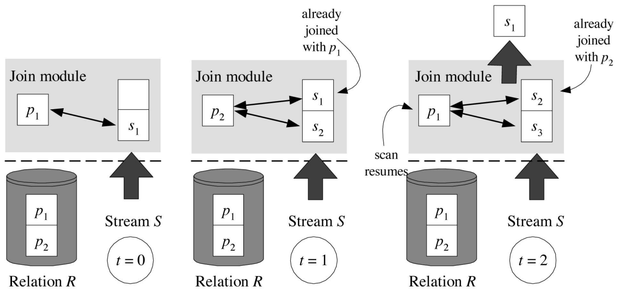


Fig. 2. Operation of MESHJOIN.

Moreover, stream tuple s_1 has been joined with all the relation and can be discarded from memory.

- At time $t = 2$, the algorithm accesses again both inputs in tandem and updates the in-memory tuples of R and S. More precisely, it resumes the scan of the relation, brings in page p_1, and simultaneously replaces tuple s_1 with the next stream tuple s_3. Page p_1 is thus joined with s_2 and s_3, and tuple s_2 is discarded as it has been joined with all the pages in R.

The previous example demonstrates the crux behind our proposed MESHJOIN algorithm: The two inputs are accessed continuously and *meshed* together in order to generate the results of the join. More precisely, MESHJOIN performs a cyclic scan of relation R and joins its tuples with a sliding window over S. The main idea is that a stream tuple enters the window when it arrives and is expired from the window after it has been probed with every tuple in R (and hence, all of its results have been computed). Fig. 3 shows a schematic diagram of this technique and depicts the main data structures used in the algorithm. As shown, MESHJOIN performs the continuous scan of R with an input buffer of b pages. To simplify our presentation, we assume that the number of pages in R is equal to $N_R = k \cdot b$ for some integer k, and hence, the scan wraps to the beginning of R after k read operations. Stream S, on the other hand, is accessed in batches of w tuples that are inserted in the contents of the sliding window. (Each insert, of course, causes the displacement of the "oldest" w tuples in the window.) To efficiently find the matching stream tuples for each R-tuple, the algorithm synchronously maintains a hash table H for the in-memory S-tuples based on their join key. Finally, queue Q contains pointers to the tuples in H and essentially records the arrival order of the batches in the current window. This information is used in order to remove the oldest w tuples from H when they are expired from the window.

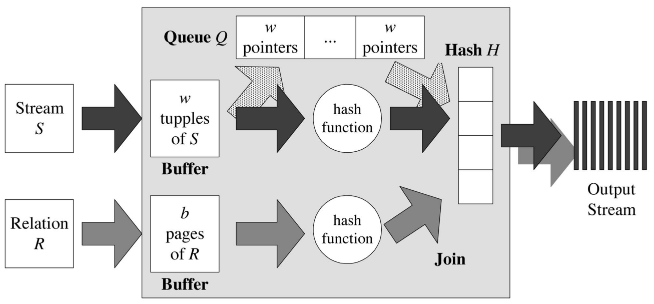


Fig. 3. Data structures and architecture of MESHJOIN.

```
Algorithm MESHJOIN
Input: A relation $R$ and a stream $S$.
Output: Stream $R \bowtie S$.
Parameters: $w$ tuples of $S$ and $b$ pages of $R$.
Method:
 1. While true
 2.     READ $b$ pages from $R$ and $w$ tuples from $S$
 3.     If queue $Q$ is full Then
 4.         DEQUEUE $T$ from $Q$ where $T$ are $w$ pointers
 5.         REMOVE the tuples of hash $H$ that correspond to $T$
 6.     EndIf
 7.     ADD the $w$ tuples of $S$ in $H$
 8.     ENQUEUE in $Q$, $w$ pointers to the above tuples in $H$
 9.     For each tuple $r$ in the $b$ pages of $R$
10.         Ouput $r \bowtie H$
11. EndWhile
```

Fig. 4. Algorithm MESHJOIN.

Fig. 4 shows the pseudocode of the MESHJOIN algorithm. On each iteration, the algorithm reads w newly arrived stream tuples and b disk pages of R, joins the R-tuples with the contents of the sliding window, and appends any results to the output buffer. The main idea is that the expensive read of the b disk pages is amortized over all the wN_R/b stream tuples in the current window, thus balancing the slow access of the relation against the fast arrival rate of the stream.

The following theorem formalizes the correctness of the algorithm. The proof appears in Appendix A.1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2008.27.

Theorem 3.1. *MESHJOIN correctly computes the exact join between a stream and a relation.*

A natural question is whether MESHJOIN requires a bounded amount of memory in order to process its streaming input and generate the correct result. In particular, this concerns the buffering of new S tuples as the algorithm is performing the current iteration. Here, we can make the following intuitive observation. For a given relation R, the service rate of MESHJOIN (essentially, the time to perform one iteration) depends on the values of w and b. (We discuss the computation of μ based on w and b later.) By treating the operator as a queuing system, we can state that the amount of required memory is bounded provided that $\mu \geq \lambda$, i.e., *the join operator can keep up with the arrival rate of the stream*. We examine this point further in the subsequent section, where we develop an analytical model for the performance of MESHJOIN and use it to tune the service rate μ based on the arrival rate λ.

3.2 Cost Model

In this section, we develop a cost model for MESHJOIN. Our cost model provides the necessary analytical tools to interrelate the following key parameters of the problem: 1) the stream rate λ, 2) the service rate μ of the join, and 3) the memory M used by the operator. Our goal will be to link these parameters to the operating parameters of MESHJOIN, namely, the number of stream tuples w that update the sliding window, and the number of pages b that can be stored in the relation buffer. The eventual goal is to employ this cost model in order to tune the parameters of

TABLE 1
Notation of the Cost Model

Parameter	Symbol
Size of tuples of S (in bytes)	v_S
Stream rate	λ
Number of pages of R	N_R
The selectivity or relation R	σ
Size of tuples of R (in bytes)	v_R
Size of a page (in bytes)	v_P
Cost of reading b pages of R	$c_{I/O}(b)$
Cost of removing a tuple from H	c_E
Cost of reading a tuple from the stream buffer	c_S
Cost of adding a tuple in H and Q	c_A
Cost of probing a hash table of size $w\frac{N_R}{b}$	c_H
Cost of creating a result tuple	c_O
Memory used by MESHJOIN	M
Total memory budget	M_{max}
Cost of a While loop of MESHJOIN	c_{loop}
Service rate of the join module	μ
Number of I/O's per tuple	IO_t
Number of I/O's per second	IO_s

the algorithm based on the characteristics of the input. The notation used in our discussion is summarized in Table 1.

The total memory M required by MESHJOIN can be computed by summing up the memory used by the buffers, the hash table H, and the queue Q. We can easily verify that

1. the buffer of R uses $b \cdot v_P$ bytes,
2. the buffer of S uses $w \cdot v_S$ bytes,
3. the queue Q uses $w \cdot \frac{N_R}{b} \cdot \text{sizeof}(ptr)$ bytes (where $\text{sizeof}(ptr)$ is the size of a pointer), and
4. the hash table H uses $w \cdot f \cdot \frac{N_R}{b} \cdot v_S$ bytes (where f is the fudge factor of the hash table implementation).

Thus, we have

$$\begin{aligned} M = b \cdot v_P + w \cdot v_S + w \cdot \frac{N_R}{b} \cdot \text{sizeof}(ptr) \\ + w \cdot f \cdot \frac{N_R}{b} \cdot v_S \leq M_{max}. \end{aligned} \tag{1}$$

The previous equation describes the effect of w and b on the memory consumed by MESHJOIN. Next, we analyze the effect of w and b on the processing speed of the operator. We use c_{loop} to denote the cost of a single iteration of the MESHJOIN algorithm and express it as the sum of costs for the individual operations. In turn, the cost of each operation is expressed in terms of w, b, and an appropriate cost factor that captures the corresponding CPU or I/O cost. These cost factors are listed in Table 1 and are straightforward to measure in an actual implementation of MESHJOIN. In total, we can express the cost c_{loop} as follows:

$$\begin{aligned} c_{loop} = c_{I/O}(b) + & \text{ (Read } b \text{ pages)} \\ w \cdot c_E + & \text{ (Expire } w \text{ tuples from } Q \text{ and } H) \\ w \cdot c_S + & \text{ (Read } w \text{ tuples from the stream buffer)} \\ w \cdot c_A + & \text{ (Add } w \text{ tuples to } Q \text{ and } H) \\ b\frac{v_P}{v_R} c_H + & \text{ (Probe } H \text{ with } R\text{-tuples)} \\ \sigma b \frac{v_P}{v_R} c_O & \text{ (Construct results).} \end{aligned} \tag{2}$$

Every c_{loop} seconds, algorithm MESHJOIN handles w tuples of the stream with b I/Os to the hard disk. Thus, the service rate μ of the join module (i.e., the number of tuples per second processed by MESHJOIN) is given by the following formula:

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

Moreover, the number of read requests per stream tuple and per time unit (denoted as IO_s and IO_t, respectively) are given by the following formulas:

$$\text{IO}_s = \frac{b}{w} \quad \text{and} \quad \text{IO}_t = \frac{b}{c_{loop}}. \tag{4}$$

The expression of IO_s demonstrates the amortization of the I/O cost over multiple stream tuples. Essentially, the cost of *sequential access* to b pages is shared among all the w tuples in the new batch, thus increasing the efficiency of accessing R. We can contrast this with the expected I/O cost of an INLs algorithm, where the index probe for each stream tuple is likely to cause at least one random I/O operation in practice. This difference is indicative of the expected benefits of our approach.

Finally, from (3) and the basic observation that $\lambda \leq \mu$, we can derive the relation between λ, c_{loop}, and w:

$$\lambda \leq \mu \Rightarrow \lambda c_{loop} \leq w. \tag{5}$$

By substituting the expression for c_{loop}, we arrive at an inequality that links w and b to the arrival rate of the stream. Combined with (1) that links w and b to the memory requirements of the operator, the previous expression forms our basic tool for tuning the parameters of MESHJOIN according to different objectives.

3.3 Tuning

We now describe the application of our cost model to the tuning of the MESHJOIN algorithm. We investigate how we can perform constrained optimization on two important objectives: minimizing the amount of required memory given a desirable service rate μ and maximizing the service rate μ, assuming that memory M is fixed. As described earlier, our goal is to achieve these optimizations by essentially modifying the parameters w and b of the algorithm. In the remainder of our discussion, we will assume that we have knowledge of the first set of parameters shown in Table 1, i.e., the physical properties of the stream and the relation, and the basic cost factors of our algorithm's operations. The former can be known exactly from the metadata of the database, while the latter can be measured with microbenchmarks.

In what follows, we discuss the details of the tuning methodology. We first examine the offline case, where the parameters of the algorithm are tuned before the start of join processing based on the predicted characteristics of the stream. We then extend our discussion to online tuning, where the algorithm continuously monitors the characteristics of the stream and adapts on the fly the parameters of the join.

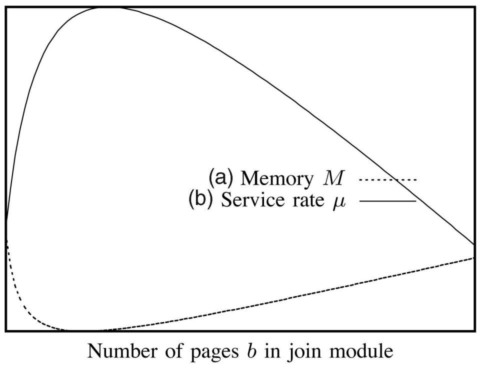


Fig. 5. (a) Minimizing M (μ is fixed), and (b) maximizing μ (M is fixed).

Offline tuning. As mentioned earlier, we consider two objectives when tuning MESHJOIN: minimizing memory consumption and maximizing the service rate.

Minimizing M. In this case, we assume that the stream rate λ is known, and we want to achieve a matching service rate $\mu = \lambda$ using the least amount of memory M. The following observations devise a simple methodology for this purpose:

1. M depends linearly on w (1). Therefore, to minimize M, we have to minimize w.
2. The minimum value for w is specified by (5) as follows: $w = \lambda c_{loop}$. This value corresponds to the state of the algorithm where the service rate of MESHJOIN is tuned to be exactly the as with the stream rate, i.e., $\lambda = \mu$.
3. The previous expression allows to solve for w and substitute the result in (1), thus specifying M as a function of b. Using standard calculus methodology, we can find exactly the value of b that minimizes M. Given (1), this also implies that we can determine a suitable value for w for the given b value.

A more intuitive view of the relationship between M and b is presented in Fig. 5a that shows M as a function of b. (Note that the curve is plotted for sample values of the cost factors obtained through simplified measurements. The actual maxima/minima of the curve may shift depending on the cost factors, but the asymptotic behavior will remain the same for realistic scenarios.) As shown, in Fig. 5a, memory consumption can vary drastically and is minimized for a specific value of b. The key intuition is that there is an inherent trade-off between b and w for maintaining a desired processing rate. For small values of b, the efficiency of I/O operations decreases as it is necessary to perform more reads of b pages in order to cover the whole relation. As a result, it is necessary to distribute the cost across a larger sliding window of stream tuples, which increases memory consumption. A larger value of b, on the other hand, allows the operator to maintain an affordable I/O cost with a small w, but the memory consumption is then dominated by the input buffer of R. We note that even though there is a single value of b that minimizes M, it is more realistic to assume that the system picks a range of b values that guarantee a reasonable behavior for memory consumption.

```
Algorithm AMESHJOIN
Input: A relation $R$ and a stream $S$.
Output: Stream $R \bowtie S$.
Parameters: $w$ tuples of $S$ and $b$ pages of $R$.
Method:
 1. $currPage = 0$
 2. While true
 3.     GET values for $w$ and $b$ from tuning module
 4.     READ pages $[currPage, currPage + b)\%N_R$ from $R$
 5.     ADD the $w$ tuples of $S$ in $H$
 6.     ENQUEUE in $Q$, $(\mathbf{w}, currPage)$ where $\mathbf{w}$ are pointers
        to the above tuples in $H$
 7.     For $i \leftarrow 1$ to $b$
 8.         JOIN in-memory page currPage with $H$
 9.         If $(currPage = head(Q).startPage - 1)\ \%N_R$
10.             DEQUEUE $head(Q)$
11.             REMOVE the tuples in $H$ that correspond to $head(Q)$
12.         EndIf
13.         $currPage \leftarrow (currPage + 1)$ mod $N_R$
14.     EndFor
15. EndWhile
```

Fig. 6. Algorithm AMESHJOIN.

Maximizing μ. Here, we assume that the available memory for the algorithm M is fixed, and we are interested in maximizing the service rate μ. Using the expressions for M, c_{loop} and μ ((1), (2), and (3), respectively), we can specify μ as a function of b and subsequently find the value that maximizes μ using standard calculus methodology.

Fig. 5b shows the derived relationship between μ and b based on the previous methodology. (We employ the same sample cost factors, as in Fig. 5a.) We observe that μ increases with b up to a certain maximum and then sharply decreases for larger values. This can be explained as follows: For small values of b, the efficiency of I/O is decreased, and the constrained memory M does not allow the effective distribution of I/O cost across many stream tuples; moreover, the cost of probing the hash table H becomes more expensive, as it records a larger number of tuples. As b gets larger, on the other hand, it is necessary to decrease w in order to stay within the given memory budget, and thus, the I/O cost per tuple increases (4). It is necessary therefore to choose the value of b (and in effect of w) that balances the efficiency of I/O and probe operations.

Online tuning. Up to this point, we have assumed that parameters w and b remain fixed for the operation of MESHJOIN and are thus tuned before the operator begins its processing. We now consider the online version of the tuning problem, where a self-tuning mechanism continuously monitors the arrival rate of the stream and dynamically adapts w and b in order to achieve an equal service rate with the least memory consumption. In this direction, we introduce an extension of the basic algorithm that can accommodate the midflight change of b. (The original algorithm can readily handle a midflight change of w provided that it does not violate the memory constraints of the problem.) We term the new algorithm AMESHJOIN and describe its details in what follows.

The pseudocode for AMESHJOIN is shown in Fig. 6. We use the notation $[a, b)\%N_R$ to denote the interval $[a, b)$ if $b < N_R$ or the interval $[a, N_R) \cup [0, b\%N_R)$ otherwise. We also use $(a = b)\%N_R$ to denote equality under modulo arithmetic. The algorithm employs a modified queue that records "packets" of stream tuples that may have different sizes. For each packet, the queue Q records an integer $startPage \in [0, N_R)$. When a packet is enqueued, $startPage$ is set to the current page counter $currPage$, and thus, $startPage$ always records the first page of R that is joined with the corresponding packet. Similar to MESHJOIN, AMESHJOIN reads concurrently from R and S in every iteration. A main difference, of course, is that the parameters b and w are set at the beginning of each iteration instead of being fixed. At the beginning of an iteration, the algorithm reads the pages in the range $[currPage, currPage + b)\%N_R$ and joins them with the current contents of H. After the join of each page $currPage + i$ $(0 \leq i < b)$, the algorithm checks whether page $currPage + i$ corresponds to the last page that needs to be joined with the first packet in Q. If so, the packet is dequeued before the algorithm proceeds with the remaining pages.

Example 3.1. Assume that relation R contains three pages $(p_0, \ldots, p_2)$ and that initially $w = 1$ and $b = 1$. In what follows, we describe the operation of AMESHJOIN at different time instants.

During the first loop $(currPage = 0)$, the algorithm reads page p_0 of R and the first stream tuple s_0. Then, the algorithm places s_0 to hash H and $(ptr(s_0), 0)$ to queue Q (where $ptr(s_0)$ is a pointer to s_0 in H). Finally, the algorithm joins p_0 and s_0 (lines 8-12) and sets $currPage = 1$. Summarizing, at the end of the first loop, we have

$currPage = 1$

Queue	Joined with pages
$(ptr(s_0), 0)$	p_0

At the second loop, AMESHJOIN algorithm reads pages p_1 and the next stream tuple s_1. Then, the algorithm places s_1 to hash H and $(ptr(s_1), 1)$ to queue Q. Finally, the algorithm joins p_1 with s_0 and s_1 and sets $currPage = 2$. Summarizing, at the end of the second loop, we have

$currPage = 2$

Queue	Joined with pages
$(ptr(s_0), 0)$	p_0,p_1
$(ptr(s_1), 1)$	p_1

Let us assume that b is set to 2 for the third loop of AMESHJOIN. This change is again initiated by the self-tuning module that monitors the arrival rate of the stream. Based on this change, the algorithm reads the next stream tuple s_2 and pages p_2 and p_0 of R (these pages correspond to interval $[currPage, currPage + b)\%N_R = [2, 4]\%N_R = [2, 3) \cup [0, 1)$ in line 4). Then, the algorithm places s_2 to hash H and $(ptr(s_2), 2)$ to queue Q and executes the For loop (lines 8-14) $b = 2$ times. In the first iteration, the AMESHJOIN algorithm joins page p_2 with s_0, s_1, and s_2. Since s_0 is joined with all relation R, $(s_0, 0)$ is dequeued from Q, and the corresponding tuples are removed from H (lines 9-12). Then, AMESHJOIN algorithm sets $currPage = 0$. Thus, we have

$currPage = 0$

Queue	Joined with
$(ptr(s_1), 1)$	p_1, p_2
$(ptr(s_2), 2)$	p_2

In the second iteration $(i = 2)$, the algorithm joins p_0 with s_1 and s_2. Then, since s_1 is joined with all relation R, the algorithm dequeues $(s_1, 1)$ from Q and removes the corresponding tuples is H (lines 9-12) Finally, AMESHJOIN sets $currPage = 1$. Thus, at the end of the third loop, we have

$currPage = 1$

Queue	Joined with
$(ptr(s_2), 2)$	p_0, p_2

The following theorem formalizes the correctness of the algorithm. (The proof appears in Appendix A.2, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2008.27.)

Theorem 3.2. *Let μ_i be the service rate achieved by AMESHJOIN in iteration $i > 0$, based on the corresponding values of b and w. AMESHJOIN computes correctly $S \bowtie R$ provided that $\mu_i \leq \lambda$, $i > 0$.*

Using the AMESHJOIN algorithm, we can device a dynamic tuning mechanism that handles the current arrival stream rate with the least memory consumption. This mechanism monitors the arrival stream rate and uses the cost model of Section 3.2 to identify the minimum memory M required to sustain the current stream rate and the respective values for b and w. (These values are used by AMESHJOIN in its next iteration.) Of course, there are cases when the system may not be able to satisfy the request for an increased memory allocation. In this case, AMESHJOIN can resort to load shedding in order to reduce the effective stream arrival rate. (We examine this topic in more detail in Section 4.)

3.4 Extensions

In this section, we discuss possible extensions of the basic MESHJOIN scheme that we introduced previously.

Ordered join output. The basic algorithm does not preserve *stream order* in the output, i.e., the resulting tuples do not necessarily have the same order as their corresponding input stream tuples. To see this, consider two consecutive stream tuples τ_s and τ'_s that have the same join key and enter the sliding window in the same batch. Assuming that $r_1, r_2, \ldots$ are the joining R-tuples for τ_s and τ'_s, it is straightforward to verify that the output stream will have the form $\ldots(\tau_s, r_1)(\tau'_s, r_1)\ldots(\tau_s, r_2)(\tau'_s, r_2)\ldots$, whereas an order-preserving output would group all the results of τ_s and τ'_s together. For all practical purposes, this situation does not compromise the correctness of the ETL transformations that we consider in our work. In those cases, where the output must observe the input stream order, it is possible to extend MESHJOIN with a simple buffering mechanism that attaches the join results to the corresponding entry in H and pushes them to the output when the tuple is dequeued and expired.

Other join conditions. MESHJOIN can be fine-tuned to work with other join conditions. The algorithm remains the

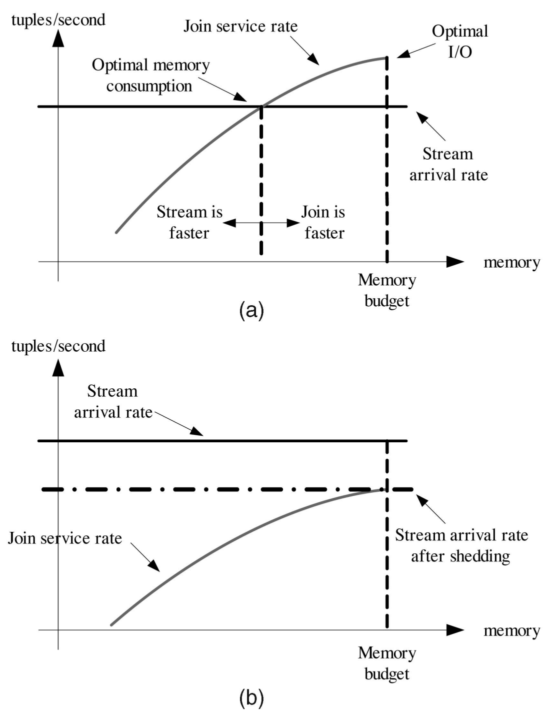


Fig. 7. (a) Memory-constrained operation. (b) Rate-constrained operation.

same, as detailed in Fig. 4, and any changes pertain mainly to the matching of R-tuples to the current contents of the sliding window (line 10). For an inequality join condition, for instance, MESHJOIN can simply buffer stream tuples in Q and process them sequentially for every accessed tuple of R. It is possible to perform further optimizations to this baseline approach depending on the semantics of the join condition. If the latter is a range predicate, for example, then the buffered stream tuples may be kept ordered to speedup the matching to R-tuples. Overall, the only requirement is to maintain the equivalent of queue Q in order to expire tuples correctly on every iteration.

4 Approximate Join Processing

Up to this point, we have focused on the case where the join operator is assigned enough memory to withstand the arrival rate of the stream (Fig. 7a). Due to the typically high number of data sources, however, it is possible to observe the converse scenario, that is, a stream arrival rate that exceeds the maximum service rate of MESHJOIN. One possible solution in this scenario is approximate query processing, which essentially trades data completeness with server resources. As an example, consider a stockbroker that monitors a report for certain stocks. The report comprises two parts: 1) the trend of the stock's price over the last week and 2) the fluctuation of the stock's price in today's auction with a freshness of 30 minutes. Moreover, assume that these values are compared to the prices of the same week in the last three years (i.e., involving a join to a persistent relation), as well as to overall trends of the stock market. The report involves a traditional query to the warehouse for values persistently stored at the appropriate fact table, along with a continuous query over the updates sent in the form of a stream by the proper source. Assuming several data sources and tens to hundreds of end users in this warehouse, the problem that arises has to do with the contention for resources between the update procedure and the end-user reports. In the case that the warehouse server cannot sustain the overall load, a typical solution is to trade the 100 percent completeness and consistency of the incoming streaming data for more resources (I/O, CPU, memory) that can be shared among the other processes.

Based on these observations, we envision a system architecture that combines approximate online query answering with batch offline refreshes. In this architecture, the ETL process has two variants: a lightweight one, which is used to provide end users with as fresh data as possible, perhaps at the cost of answer accuracy, and a regular one that synchronizes the stored data at the warehouse in order to guarantee 100 percent completeness and consistency with respect to the sources. (This can be achieved either offline or at periods with low user workload and possibly requires some extra synchronization with the sources as traditionally happens in ETL processes.)

The lightweight ETL process relies on tuple-shedding, i.e., dropping a fraction of the incoming stream tuples,[1] in order to match the speed of the join operators to the arrival rate of the stream. This is shown pictorially in Fig. 7b that illustrates the relationship among the original arrival rate, the effective arrival rate (after shedding), and the service rate. According to the system architecture described earlier, the data that is shed is stored and propagated to the data warehouse in a later time through the regular ETL process.

In general, the shedding mechanism has to take into account the details of the complete workflow in order to minimize the effect of shedding on the quality of the approximate results. A variant of this problem has been explored in the context of streaming data, where the ETL workflow essentially consists of relational operators. The problem becomes substantially more complex in our context due to the generality of ETL operations. Here, we examine different shedding mechanisms under the assumption that MESHJOIN is the most expensive operation in the workflow, and the shedder is thus driven by the cost model of the MESHJOIN operator. The design of shedding strategies for complex (and active) ETL workflows is an interesting topic for future work.

4.1 Overview of Problem Space

Formally, let λ be the current arrival rate of the stream and μ the current service rate of the algorithm. Based on the cost model of Section 3.1, this service rate is determined by the cost c_{loop} of a single iteration and the block size w of stream tuples that is processed. Hence, a fast stream will deposit $c_{loop}\lambda > w$ new stream tuples at the end of each iteration, implying that the join algorithm will fall behind at a rate of $\lambda - w/c_{loop}$ tuples per second. The goal of the shedding

1. A dual strategy is to process less R tuples on each iteration of MESHJOIN, thus reducing the effective size of the disk-based input. Nevertheless, a third (hybrid) strategy can shed tuples from both inputs. In this work, we only consider the solution that is based on stream shedding. Exploring the other strategies and their trade-offs is an interesting topic for future work.

TABLE 2
Stream Shedding Strategies

	MAX-SUBSET	**RANDOM-SAMPLE**
Many-to-One	*Keep w*	*Sample w*
Many-to-Many	TOPW	Cluster Sampling [7]

mechanism, therefore, is to keep a fraction $w/(\lambda c_{loop})$ of the incoming stream tuples in order to allow MESHJOIN to keep up. The decision of which tuples to keep depends, of course, on the specifics of the strategy.

Following previous studies on approximate join processing [6], [7], we consider two criteria to characterize the loss of result tuples: MAX-SUBSET and RANDOM-SAMPLE. In short, MAX-SUBSET attempts to maximize the subset of generated results or, alternatively, to minimize the number of lost result tuples. RANDOM-SAMPLE generates a random sample of the join output. Previous studies have explored these objectives in the context of window-based joins for two streaming inputs. Our problem is substantially different as we deal with one stream, we do not impose a window constraint, and we do not assume that there is enough memory to hold the disk-based relation.

In addition to the loss criterion, we introduce a second parameter related to the join relationship between S and R. More concretely, we distinguish between a *many-to-one* join, where each stream tuple joins with exactly one relation tuple, and a *many-to-many* join. This distinction is motivated by the applications of MESHJOIN in practice, where a stream is likely to carry a foreign key on the static relation. As we discuss below, we can devise very efficient shedding strategies for this common case.

Table 2 summarizes our proposed shedding strategies for the space of problem parameters:

- *Many-to-One/MAX-SUBSET.* The optimal shedding strategy is to simply maintain w stream tuples out of the λc_{loop} tuples that arrive during one iteration of MESHJOIN. Given that each dropped stream tuple corresponds to exactly one result tuple, this strategy guarantees the minimal loss of $\lambda c_{loop} - w$ result tuples.
- *Many-to-One/RANDOM-SAMPLE.* In this case, it suffices to maintain a random uniform sample of size w out of the λc_{loop} tuples. The key/foreign-key constraint guarantees that the result tuples form a random uniform sample of the join result, with a sampling rate of $w/(\lambda c_{loop})$ [8].
- *Many-to-Many/MAX-SUBSET.* We propose a shedding heuristic, termed TOPW, for dropping tuples of low frequency. The details of the heuristic are presented below.
- *Many-to-Many/RANDOM-SAMPLE.* Following an earlier study [7] on approximate join processing over streams, we adopt Cluster Sampling as the shedding strategy. Briefly, this entails sampling a stream tuple with probability proportional to its join frequency, i.e., the number of tuples in R with the same join value. Even though this does not yield a random uniform sample of the result, it still permits the computation of bounds on aggregates computed over the join [7].

Clearly, the design of the MESHJOIN algorithm enables very effective and straightforward solutions for the case of many-to-one joins. For this type of joins, both shedding strategies have strong guarantees on their properties: *Keep w* will return a maximal subset of the join results, while *Sample w* will generate a random uniform sample with a maximal sampling rate. Hence, MESHJOIN can handle effectively the case of many-to-one joins (essentially, key/foreign-key joins) that are predominant in real-world applications.

4.2 Shedding Strategies for Many-to-Many/MAX-SUBSET

In what follows, we analyze shedding strategies suitable for the subspace Many-to-Many/MAX-SUBSET. This analysis is a contribution of our work, since existing strategies do not extend in this setting. We introduce an online shedding strategy, termed TOPW, that does not require a priori knowledge of the stream and is thus applicable in any scenario. To quantify the performance of TOPW, we analyze an offline shedding strategy, termed OPTOFFLINESHED, that has knowledge of the complete stream and achieves an optimal loss of join results. We note that this strategy is clearly infeasible in practice, and the intention behind its introduction is solely to define a benchmark for online strategies.

Before proceeding with our presentation, we introduce some necessary notation. We treat the stream as a finite sequence $s_0, s_1, \ldots, s_{|S|}$, where each entry s_i has a time stamp t_i and an associated stream tuple τ_i. (The switch to finite streams is necessary so that an *offline* strategy becomes meaningful. Clearly, the *online* TOPW heuristic can be applied on infinite streams.) For each stream tuple τ_i, we will use $f_R(i)$ to denote the count of R-tuples that join with τ_i. Essentially, $f_R(i)$ can be derived from the distribution of values in the join attributes of R. We note that such statistics are already maintained by the query optimizer, as they are necessary for query compilation. It is thus safe to assume that $f_R(i)$ is readily available without additional overhead.

Online shedding—Algorithm TOPW. Conceptually, TOPW operates in parallel with MESHJOIN and maintains a buffer of w tuples that is transferred to the join module whenever requested. As the name suggests, the algorithm maintains the top w tuples of S according to their matching frequency f_R in R. The intuition is to maximize the number of join results. More formally, let tr be the time stamp of the last transfer of w tuples between TOPW and MESHJOIN. TOPW maintains a max heap of size w for the tuples that arrive in the interval $[tr, tr + c_{loop}]$ ordered by their frequency values $f_R(i)$. Hence, MESHJOIN receives the w tuples from the interval $[tr, tr + c_{loop}]$ that join with the most tuples in R. The link between our heuristic and MAX-SUBSET is evident.

Clearly, TOPW is a heuristic algorithm and can thus miss the optimal shedding strategy. Fig. 8 illustrates this case with a sample stream of eight elements. Assume that $w = 2$ and $c_{loop} = 4$ time units. For each stream element, the figure shows the corresponding frequency f_R, i.e., how

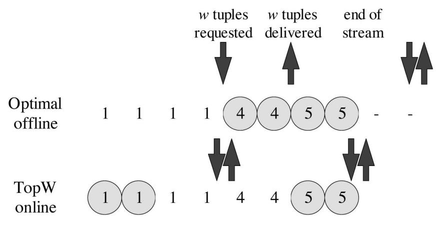


Fig. 8. TOPW versus optimal strategy.

many join tuples will be lost if the stream tuple is dropped. The arrows indicate the points where a request is made by MESHJOIN (downward pointing arrow) and when the tuples are transferred (upward pointing arrow), and the returned tuples are enclosed in circles. As shown, TOPW always maintains the top w tuples in the current interval and returns them as soon as they are requested, achieving a total loss of 10 tuples. In this example, however, the optimal strategy is to stall MESHJOIN in order to include the last stream tuples in the join, achieving a total loss of 4 tuples. As the example shows, the performance of TOPW is likely to suffer if the stream tuples that generate a high number of results are clustered in the time dimension. In practice, we may expect a more random distribution of losses within the stream, and hence, our TOPW heuristic is likely to perform well.

Offline optimal shedding. We now present an offline shedding algorithm, termed OPTOFFLINESHED, that assumes a priori knowledge of the incoming stream tuples and can thus determine an optimal shedding strategy for the MAX-SUBSET objective. Clearly, this algorithm cannot be implemented in practice and only serves as the benchmark for our online heuristic.

Similar to TOPW, OPTOFFLINESHED maintains a working buffer of w tuples that is consumed by MESHJOIN. As Fig. 8 suggests, however, a key difference is that OPTOFFLINESHED will have the option of stalling the join operator. More precisely, we assume that MESHJOIN places a request for the next packet of w tuples at the end of its current iteration, and this request is satisfied by OPTOFFLINESHED at some point in the future (but not necessarily immediately). For ease of exposition, we assume that time stamps take integer values (with $t_0 = 0$) and that c_{loop} is also an integer. Moreover, we assume that time stamps are consecutive, i.e., $t_i = t_{i-1} + 1 = i$ (since $t_0 = 0$). This assumption does not compromise the generality of our analysis, as any stream can be extended with zero-effect tuples (i.e., $f_R(i) = 0$) in the missing time stamps. Finally, we assume that OPTOFFLINESHED does not substitute tuples in the working buffer, i.e., once a tuple is stored in the buffer, then it will be eventually processed by MESHJOIN. Again, this does not restrict the applicability of our analysis, as a tuple substitution is equivalent to a strategy that does not select the specific tuple in the first place.

Our main observation is that the optimal selection of tuples from time i onward depends only on the available buffer space and not on the actual tuples that have been placed in the buffer. This suggests a dynamic-programming approach to computing the optimal solution. More formally, let $optloss(i, lt, sb)$ denote the optimal loss assuming that 1) the algorithm has to select tuples from s_i onward, 2) MESHJOIN will make its next request for tuples at time stamp lt, and 3) the algorithm has already selected $sb \le w$ tuples in the buffer. Hence, $optloss(0, 0, 0)$ denotes the optimal loss for the whole stream. For completeness, we assume that $optloss(i, lt, sb) = 0$ if $i > |S|$.

```
Algorithm OPTOFFLINESHED
Input: A stream $S$; size $w$ of working buffer; cost $c_{loop}$ of a single MESHJOIN iteration.
Output: The optimal loss of any on-line shedding strategy on $S$
Method:
 1. Initialize tables $T[0..|S|+1, 0..w]$ and $T'[0..|S|+1, 0..w]$
 2. $T'[lt, sb] \leftarrow 0$ for all $0 \le lt \le |S|+1$, $0 \le sb \le w$
    // Main iteration
 3. For $i \leftarrow |S|$ to 0
    // First case: $sb < w$
 4.     For $lt \leftarrow 0$ to $\min(i + c_{loop}, |S|)$, $sb \leftarrow 0$ to $\min(i+1, w)$
 5.         $T[lt, sb] \leftarrow \min(T'[lt, sb] + f_R(i), T'[lt, sb+1])$
 6.     EndFor
    // Second case: $sb = w \wedge\ i < lt$
 7.     For $lt \leftarrow i+1$ to $\min(i + c_{loop}, |S|)$
 8.         $T[lt, w] \leftarrow T'[lt, w]$
 9.     EndFor
    // Third case: $sb = w \wedge\ i \ge lt$
10.     For $lt \leftarrow 0$ to $i$
11.         $T[lt, w] \leftarrow T[i + c_{loop}, 0]$
12.     EndFor
13.     Swap $T$ and $T'$
14. EndFor
15. Return $T'[0, 0]$
```

Fig. 9. Algorithm OPTOFFLINESHED.

We start the analysis with the case $sb < w$. Obviously, the algorithm can store s_i in the buffer and increase the occupancy to $sb + 1$, in which case, the loss is equal to the loss from that point onward. If it chooses to exclude s_i, then the loss is $f_R(i)$ join results plus the optimal loss from $i + 1$ onward under the same buffer occupancy. Clearly, the optimal choice is the minimum of the two options, and this leads to the following expression:

$$\begin{aligned} optloss(i, lt, sb) = \min(&optloss(i+1, lt, sb+1),\\ &optloss(i+1, lt, sb) + f_R(i)). \end{aligned}$$

Next, we consider the case $sb = w$. If $i < lt$, then the algorithm cannot transfer the buffer to MESHJOIN since the latter has not requested the next batch of stream tuples. Thus, the only choice is to drop tuple s_i at a loss of $f_R(i)$. If $i \ge lt$, on the other hand, then the buffer is transferred, and the selection of tuples starts afresh with an empty working buffer and a next request time of $i + c_{loop}$. Formally, we can describe these two cases as follows:

$$optloss(i, lt, w) = \begin{cases} optloss(i+1, lt, w) + f_R(i) & i < lt, \\ optloss(i, i + c_{loop}, 0) & i \ge lt. \end{cases}$$

The previous equations form the basis behind the dynamic programming algorithm for computing the optimal loss for the particular stream. The pseudocode for the algorithm is shown in Fig. 9. The algorithm iterates over all values of i and maintains two tables, namely, $T[0..|S| + 1, 0 \ldots w]$ and $T'[0..|S| + 1, 0 \ldots w]$, that store the entries for $optloss(i, lt, sb)$ and $optloss(i + 1, lt, sb)$, respectively, for the specific i. The entries in T are computed from the entries in T'

using the previously described recurrence expressions, and the optimal loss $optloss(0,0,0)$ can be retrieved from $T'[0,0]$ at the end of the iteration. To optimize the computation, the algorithm takes into account that the next transfer request can have a maximum value of $i + c_{loop}$ relative to the current iteration and also that the number sb of stored tuples can never exceed the number of tuples that have been observed in the stream. We note that the algorithm computes $optloss(0,0,0)$ quite efficiently, with a space complexity of $O(|S|w)$ and a time complexity of $O(|S|^2 w)$.

The following theorem formalizes the optimality of OPTOFFLINESHED.

Theorem 4.1. *Algorithm OPTOFFLINESHED computes the optimal loss for any online shedding strategy in Many-to-Many/MAX-SUBSET that maintains a working buffer of w tuples and achieves an effective arrival rate of w/c_{loop}.*

5 EXPERIMENTS

In this section, we present an experimental study that demonstrates the effectiveness of our techniques. Overall, our results verify the efficacy of MESHJOIN in computing $S \bowtie R$ in the context of active ETL transformations, and demonstrate its numerous benefits over conventional join algorithms.

5.1 Methodology

The following paragraphs describe the major components of our experimental methodology, namely, the techniques that we consider, the data sets, and the evaluation metrics.

Join processing techniques. We consider two join processing techniques in our experiments.

MESHJOIN. We have completed a prototype implementation of the MESHJOIN algorithm that we describe in this article. We have used our prototype to measure the cost factors of the analytical cost model (Section 3.1). In turn, we have used this fitted cost model in order to set b and w accordingly for each experiment. We stress that the particular cost factors are inherently tied to the specific (software and hardware) characteristics of the experimental platform and are also affected by our measuring methodology. Thus, their specific values are of limited importance. Our goal instead is to demonstrate that it is possible to fit the abstract cost expressions of Section 3.1 on a specific platform in order to model the performance of MESHJOIN with reasonable accuracy.

Index-Nested-Loops. We have implemented a join module based on the INLs algorithm. We have chosen INL as it is readily applicable to the particular problem without requiring any modifications. Our implementation examines each update tuple in sequence and uses a clustered B+-Tree index on the join attribute of R in order to locate the matching tuples. We have used the Berkeley DB library (version 4.3.29) for creating and probing the disk-based clustered index. In all experiments, the buffer pool size of Berkeley DB was set equal to the amount of memory allocated to MESHJOIN.

In both cases, our implementation reads in memory the whole stream before the join starts and provides update tuples to the operator as soon as they are requested. This allows an accurate measurement of the maximum processing speed of each algorithm, as new stream tuples are accessed with essentially negligible overhead.

TABLE 3
Data Set Characteristics

Parameter	Value
z_R: skew of join attribute in R	0-1
z_S: skew of join attribute in S	0-1
D: domain of join attribute	$[1, 3.5\ 10^6]$
v_R: size of R-tuple	120 bytes
n_R: number of tuples in R	3.5M
v_S: size of S-tuple	20 bytes

Data sets. We evaluate the performance of join algorithms on synthetic and real-life data of varying characteristics.

- *Synthetic data set.* Table 3 summarizes the characteristics of the synthetic data sets that we use in our experiments. We assume that R joins with S on a single integer-typed attribute, with join values following a zipfian distribution in both inputs. We vary the skew in R and S independently and allow it to range from 0 (uniform join values) to 1 (skewed join values). In all cases, we ensure that the memory parameter M_{max} does not exceed 10 percent of the size of R, thus modeling a realistic ETL scenario, where R is much larger than the available main memory. We note that we have performed a limited set of experiments with a bigger relation of 10 million tuples, and our results have been qualitatively the same as for the smaller relation.
- *Real-life data set.* Our real-life data set is based on weather sensor data that measure cloud cover over different parts of the globe [9]. This data set encodes a sequence of records that evolve over time and is thus well suited for representing the streams of updates from the operational sources. As a more concrete example, we can view the weather sensors as the operational data sources that push their updates to a scientific data warehouse, which in turn performs a mesh join for the assignment of surrogate keys before the storage of the sensor observations. We use measurements from two different months to create a relation and a stream of update tuples, respectively. The tuple-size is 32 bytes for both R and S, and the underlying value domain is [0, 36,000]. Each input comprises 10 million tuples.

Evaluation metrics. We evaluate the performance of a join algorithm based on its service rate μ, that is, the maximum number of update tuples per second that are joined with the disk-based relation. For MESHJOIN, we let the algorithm perform the first four complete loops over relation R and then measure the rate for the stream tuples that correspond to the last loop only. For INL, we process a prefix of 100,000 stream tuples and measure the service rate on the last 10,000 tuples.

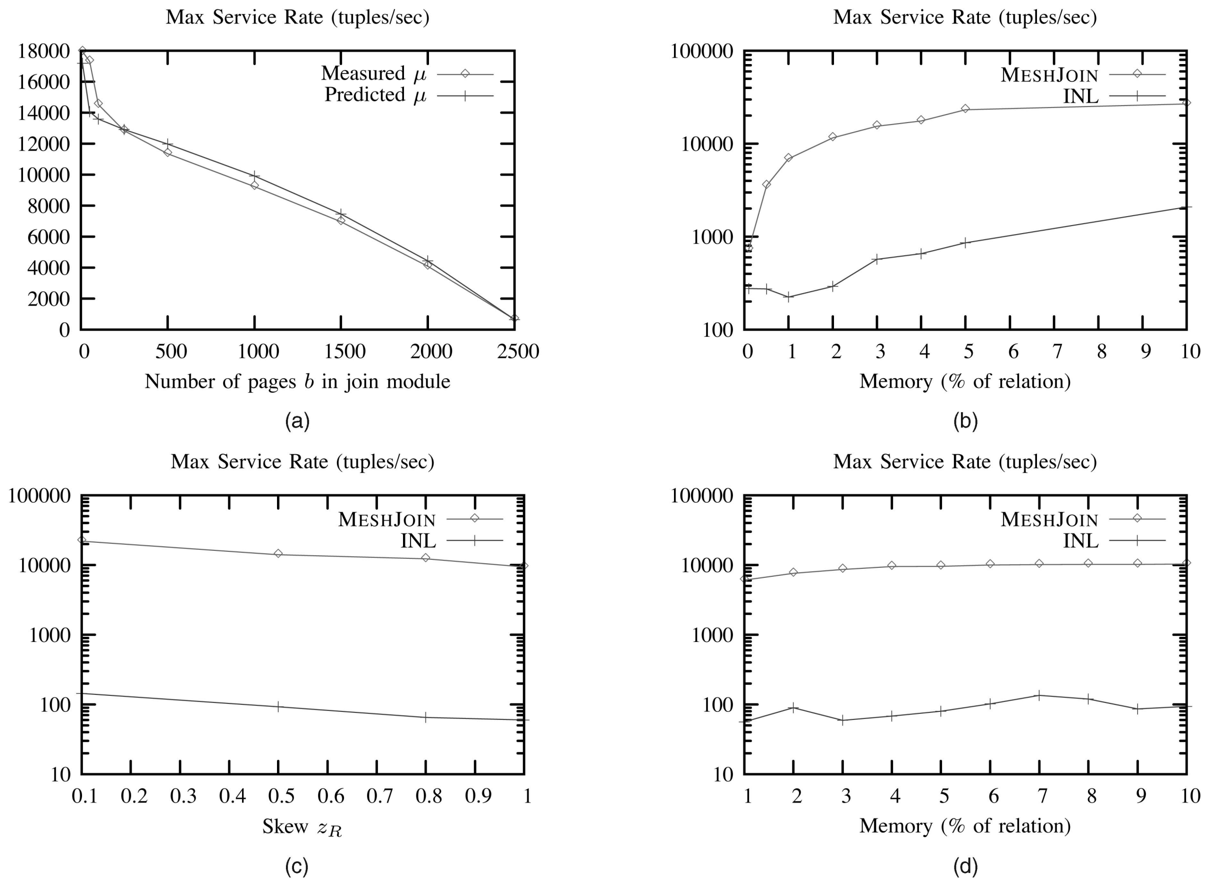


Fig. 10. Experimental evaluation of MESHJOIN. (a) MESHJOIN: predicted and measured performance (synthetic data). (b) MESHJOIN and INL: performance for varying memory (synthetic data). (c) MESHJOIN and INL: performance for varying data skew (synthetic data). (d) MESHJOIN and INL: performance for varying memory (real-life data).

Experimental platform. We have performed our experiments on a Pentium IV 3-GHz machine with 1 Gbyte of main memory running Linux. Our disk-based relations are stored on a local 7200RPM disk, and the machine has been otherwise unloaded during each experiment. In all experiments, we have ensured that the file system cache is kept to a minimum in order to eliminate the effect of double-buffering in our measurements.

5.2 Experimental Results

In this section, we report the major findings from our experimental study. We present results on the following experiments: a validation of the cost model for MESHJOIN, a sensitivity analysis of the performance of MESHJOIN, and an evaluation of MESHJOIN on real-life data sets.

Cost model validation. In this experiment, we validate the MESHJOIN cost model that we have presented in Section 3.1. We use the synthetic data set with a fixed memory budget of 21 Mbytes (5 percent of the relation size), and we vary b and w so that the total memory stays within the budget. For each combination, we measure the service rate of MESHJOIN, and we compare it against the predicted rate from the cost model.

Fig. 10a depicts the predicted and measured service rate of MESHJOIN as a function of b. (Note that each b corresponds to a unique setting for w according to the allotted memory of 21 Mbytes.) As the results demonstrate, our cost model tracks accurately the measured service rate and can thus be useful in predicting the performance of MESHJOIN. The measurements also indicate that the service rate of MESHJOIN remains consistently high for small values of b and drops rapidly as b is increased. (Our experiments with different memory budgets have exhibited a similar trend.) In essence, a large b reduces w (and effectively, the size of the sliding window over S), which in turn decreases significantly the effectiveness of amortizing I/O operations across stream tuples. This leads to an increased iteration cost c_{loop} and inevitably to a reduced service rate.

Sensitivity analysis. In this set of experiments, we examine the performance of MESHJOIN when we vary two parameters of interest, namely, the available memory budget M and the skew of the join attribute. We use synthetic data sets, and we compare the service rate of MESHJOIN to the baseline INL algorithm.

Varying M. We first evaluate the performance of MESHJOIN when we vary the available memory budget M. We assume that the join attribute is a key of the relation and set $z_s = 0.5$ for generating join values in the stream. (Our results with different skew values have been qualitatively the same.) These parameters model the generation of surrogate keys, a common operation in data warehousing. In the experiments that we present, we vary M as a percentage of the size of the disk-based relation, from 0.1 percent ($M = 200$ KB) up to 10 percent ($M = 40$ Mbytes). All reported measurements are with a cold cache.

Fig. 10b shows the maximum service rate (tuples/second) of MESHJOIN and INL as a function of the memory allocation M. Note that the y-axis (maximum service rate) is in log scale. The results demonstrate that MESHJOIN is very effective in joining a fast update stream with a slow disk-based relation. For a total memory allocation of 4 Mbytes (1 percent of the total relation size), for instance, MESHJOIN can process a little more than 6,000 tuples per second, and scales up to 26,000 tuples/second if more memory is available. It is interesting to note a trend of diminishing returns as MESHJOIN is given more memory. Essentially, the larger memory allocation leads to a larger stream window that increases the cost factors corresponding to the expiration of tuples and the maintenance of the hash table H.

Compared to INL, MESHJOIN is the clear winner as it achieves a 10 times improvement for all memory allocations. For an allotted memory M of 2 Mbytes (0.5 percent of the total relation size), for instance, INL can sustain 274 tuples/second, while MESHJOIN achieves a service rate of 3,500 tuples/second. In essence, the buffer pool of INL is not large enough to "absorb" the large number of random I/Os that are incurred by index probes, and hence, the dominant factor becomes the overhead of the "slow" disk. (This is also evident from the instability of our measurements for small allocation percentages.) MESHJOIN, on the other hand, performs continuous sequential scans over R and amortizes the cost of accessing the disk across a large number of stream tuples. As the results demonstrate, this approach is very effective in achieving high servicing rates even for small memory allocations.

Varying skew. The second set of experiments measures the performance of MESHJOIN for different values of the relation skew parameter z_R. Recall that z_R controls the distribution of values in the join column of R and hence affects the selectivity of the join. We keep the skew of the stream fixed at $z_S = 0.5$ and vary z_R from 0.1 (almost uniform) to 1 (highly skewed) for a join domain of 3.5 million values. In all experiments, the join algorithms are assigned 20 Mbytes of main memory (5 percent of the size of R). We note that we have also experimented with different values for z_S, and we have found that both techniques are relatively insensitive to this parameter.

Fig. 10c depicts the maximum service rate for MESHJOIN and INL as a function of the relation skew z_R. (Again, the y-axis is in log scale.) Overall, our results indicate a decreasing trend in the maximum service rate for both algorithms as the skew becomes higher. In the case of MESHJOIN, the overhead stems from the uneven probing of the hash table, as more R-tuples probe the buckets that contain the majority of stream tuples. (Recall that the stream is also skewed with $z_S = 0.5$.) For INL, the overhead comes mainly from the additional I/O of accessing long overflow chains in the leaves of the B+-Tree when z_R increases. Despite this trend, our proposed MESHJOIN algorithm maintains a consistently high service rate for all skew values, offering a significant performance improvement compared to INL.

Performance of MESHJOIN on real-life data sets. As a final experiment for the case of exact join processing, we present an evaluation of MESHJOIN on our real-life data set. We vary the memory budget M as a percentage of the relation size, from 1 percent (4 Mbytes) to 10 percent (40 Mbytes). Again, we compare MESHJOIN to INL using the service rate as the evaluation metric.

Fig. 10d depicts the service rate of MESHJOIN and INL on the real-life data set as a function of the memory budget. Similar to our experiments on synthetic data, MESHJOIN achieves high service rates and outperforms INL by a large margin. Moreover, this consistently good performance comes for low memory allocations that represent a small fraction of the total size of the relation.

Approximate join processing. In this set of experiments, we focus on the case of approximate join processing, i.e., when the memory budget M is not sufficient for the arrival rate of the stream. Among our shedding strategies (Section 4), we note that *Keep* w and *Sample* w have strong guarantees that are trivial to validate experimentally, while the trade-offs of Cluster Sampling have already been evaluated in a previous study [7]. Hence, we consider only TOPW in the following experiments.

We compare the performance of TOPW against the optimal offline algorithm OPTOFFLINESHED for a many-to-many join. Since we focus on the MAX-SUBSET criterion, we use the fraction of generated result tuples as the evaluation metric. We keep the memory M used by MESHJOIN fixed and vary the ratio $\rho = \lambda/\mu$ of the steam rate λ and the service rate μ of the algorithm. In our experiments, we report results for $1.1 \leq \rho \leq 2$, with $\rho = 2$, indicating a stream that is twice as fast as the service rate of the join. For a specific ρ and shedding algorithm, we measure the average fraction of join results over 10 different streams, each containing $\rho \cdot 20{,}000$ tuples with skew $z_S = 0.5$. For the disk-based data, we consider three different relations of skew z_R 0.1 (almost uniform), 0.5, and 1.0 (skewed), respectively.

As a final detail, we note that our implementation of TOPW employs accurate information on the distribution of join values in R, i.e., the information that enables TOPW to compute $f_R(i)$ for each stream tuple τ_i. We opt for this experimental design as it isolates the logic of the shedding strategy from the error of inaccurate data statistics. Thus, it becomes easier to evaluate the potential of the core idea behind the TOPW heuristic.

Fig. 11 depicts the fraction of generated join results for TOPW as a function of the speed ratio ρ, for the three relations of different skew values. (We omit the results of OPTOFFLINESHED from the plot as the two curves were essentially identical.) As shown, TOPW is effective in generating a sizeable fraction of the join results for fast incoming streams. For a relation of moderate skew $z_R = 0.5$, for instance, TOPW generates more than 80 percent of the complete join results even when the stream is two times faster than the maximum service rate. The performance of TOPW generally increases with the skew in the disk-based relation, as it always manages to select the few stream tuples that generate the most join results; a decreased skew,

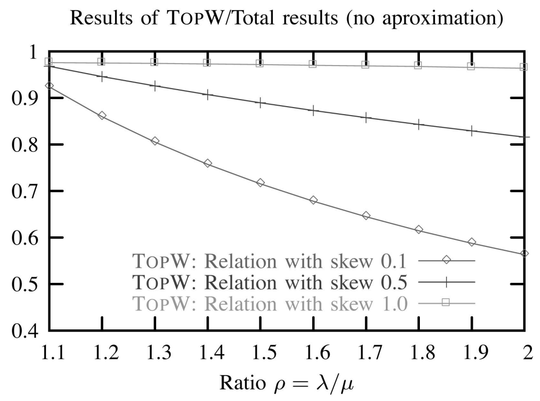


Fig. 11. Performance of load shedding.

on the other hand, implies equal contributions from all stream tuples and hence bigger losses. The important observation, however, is that the performance of TOPW, with respect to the MAX-SUBSET metric, is comparable to the optimal offline shedding algorithm.

6 Related Work

In this section, we broadly discuss research efforts regarding join algorithms and active or real-time data warehousing.

Join algorithms. Join algorithms have been studied extensively since the early days of database development, and earlier works have introduced a host of efficient techniques for the case of finite disk-based relations. (A review can be found in [10].) In recent years, the case of continuous data streams has gained in popularity, and researchers have examined techniques and issues for join processing over streaming relations [11], [12], [13], [14]. In this setting, which is more relevant to our work, we distinguish two cases for our discussion: *unbounded streams*, where a stream is modeled as an infinite relation, and *finite streams*, where a bounded relation is essentially streamed from a remote data source.

Join processing over unbounded streams. Earlier studies [15], [16], [17], [18] have introduced generalizations of Symmetric Hash-Join to a multiway join operator in order to efficiently handle join queries over multiple unbounded streams. These works, assume the application of window operators (time or tuple based) over the streaming inputs, thus reducing each stream to a finite evolving tuple-set that fits entirely in main memory. This important assumption does not apply to our problem, where the working memory is assumed to be much smaller than the large disk-based relation, and there is no window restriction on the streaming input.

To cope with fast streams that exceed the processing capacity of the join algorithm, earlier studies have considered techniques for *approximate join processing*. A commonly used approximation method is *load shedding* [19], [20], [21], where the join algorithm selectively drops input tuples in order to reduce the effective stream arrival rate. Two recent studies [6], [7] have introduced formal models for load shedding and have proposed the MAX-SUBSET and RANDOM-SAMPLE criteria. Earlier works have also considered the use of single-pass synopses in order to approximate fast streams and essentially generate an approximate join result. Examples in this category include randomized sketching techniques [22], [23], histogram synopses [24], as well as sampling-based approaches [25], [26]. Our proposed approximation strategy is related to the load-shedding models proposed in [6], [7], as we consider the same high-level strategies (MAX-SUBSET and RANDOM SAMPLE). The constraints of our solution are different, however, since we assume a different join-processing model.

Join processing over finite streams. The works in this category consider the join of finite relations that are streamed over an unstable network with temporary failures. Representative techniques are the Symmetric Hash-Join [27] and its XJoin variant [28], the Progressive Merge Join (PMJ) [29], and the more recent Rate-based Progressive Join (RPJ) [30]. At an abstract level, the proposed techniques use the following processing model: the join algorithm accesses the streaming inputs continuously and maintains the received tuples in memory in order to generate results as early as possible; when the received input exceeds the capacity of the main memory, the algorithm flushes a subset of the data to disk and processes it later when (CPU or memory) resources allow it. This model, however, is not well suited for our setting, where the stream input is possibly infinite, the access to the relation is predictable, and the requirement for online result generation favors the use of approximate join processing, rather than disk buffering, when the stream arrival rate exceeds the capacity of the join operator. The aforementioned join operators can be used of course to realize the regular ETL process and can thus complement the approximate join processing of MESHJOIN.

Novelty of our approach with respect to existing join algorithms. As noted earlier, previous studies on stream query processing have focused on the case of two finite inputs (relations) or two infinite inputs (streams) with a window predicate. Our work, on the other hand, focuses on the mixed case of joining a finite relation with an infinite stream. Even though previous studies [25], [31] have recognized this problem as an important issue in streaming database systems, to the best of our knowledge, there has been no proposal of a specialized join algorithm.

Compared to the existing INLs approach, MESHJOIN is more efficient as it relies solely on fast sequential scans in order to access disk-resident data. As our experimental study has demonstrated, this approach results in increased I/O efficiency and essentially enables MESHJOIN to scale to very fast streams under limited memory resources.

Active DW's. Active or real-time data warehousing has recently appeared in the industrial literature [1], [3], [4]. Research in ETL has provided algorithms for specific tasks, including the detection of duplicates, the resumption from failure, and the incremental loading of the warehouse [32], [33], [34]. Contrary to our setting, these algorithms are designed to operate in a batch offline fashion. Work in materialized views refreshment [35], [36], [37], [38] is also relevant but orthogonal to our setting. The crucial decision concerns whether a view can be updated given a delta set of updates. An interesting, but still orthogonal to our case, situation involves the case of concurrency conflicts during data warehouse maintenance. As we have already discussed (see Section 2), we consider that the stream is not affected by changes in other streams of updates coming from other sources. Any transactional conflict can be resolved by well-known synchronization policies outside the join module [5], [39].

7 Conclusions

In this work, we have considered an operation that is commonly encountered in the context of active data warehousing: the join between a fast stream of source updates S and a disk-based relation R under the constraint of limited memory. We have proposed the *mesh join* (MESHJOIN), a novel join operator that operates under minimum assumptions for the stream and the relation. We have developed a systematic cost model and tuning methodology that accurately associates memory consumption with the incoming stream rate. Finally, we have validated our proposal through an experimental study that has demonstrated its scalability to fast streams and large relations under a limited main memory.

References

[1] D. Burleson, *New Developments in Oracle Data Warehousing.* Burleson Consulting, Apr. 2004.

[2] A. Karakasidis, P. Vassiliadis, and E. Pitoura, "ETL Queues for Active Data Warehousing," *Proc. Second Int'l Workshop Information Quality in Information Systems (IQIS),* 2005.

[3] "On-Time Data Warehousing with Oracle10g—Information at the Speed of Your Business," white paper, Oracle Corp., Aug. 2003.

[4] C. White, "Intelligent Business Strategies: Real-Time Data Warehousing Heats Up," *DM Rev.,* 2002.

[5] S. Chen, J. Chen, X. Zhang, and E.A. Rundensteiner, "Detection and Correction of Conflicting Source Updates for View Maintenance," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '04),* pp. 436-448, 2004.

[6] A. Das, J. Gehrke, and M. Riedewald, "Approximate Join Processing over Data Streams," *Proc. ACM SIGMOD,* 2003.

[7] U. Srivastava and J. Widom, "Memory-Limited Execution of Windowed Stream Joins," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2004.

[8] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy, "Join Synopses for Approximate Query Answering," *Proc. ACM SIGMOD,* 1999.

[9] C.J. Hahn, S.G. Warren, and J. London, "Edited Synoptic Cloud Reports from Ships and Land Stations over the Globe, 1982-1991," http://cdiac.ornl.gov/epubs/ndp/ndp026b/ndp026b.htm, 2007.

[10] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys,* vol. 25, no. 2, 1993.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," *Proc. ACM Symp. Principles of Database Systems (PODS),* 2002.

[12] S. Babu and J. Widom, "Continuous Queries over Data Streams," *SIGMOD Record,* vol. 30, no. 3, 2001.

[13] L. Golab and M. Tamer Özsu, "Issues in Data Stream Management," *SIGMOD Record,* vol. 32, no. 2, 2003.

[14] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous Queries over Append-Only Databases," *Proc. ACM SIGMOD,* 1992.

[15] S. Chandrasekaran and M.J. Franklin, "PSoup: A System for Streaming Queries over Streaming Data," *Very Large Data Bases J.,* vol. 12, no. 2, 2003.

[16] L. Golab and M. Tamer Özsu, "Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2003.

[17] M. Hammad, M.J. Franklin, W. Aref, and A. Elmagarmid, "Scheduling for Shared Window Joins over Data Streams," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2003.

[18] S. Viglas, J.F. Naughton, and J. Burger, "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2003.

[19] B. Babcock, M. Datar, and R. Motwani, "Load-Shedding for Aggregation Queries over Data Stream," *Proc. IEEE Int'l Conf. Data Eng. (ICDE),* 2004.

[20] J. Kang, J. Naughton, and S. Viglas, "Evaluating Window Joins over Unbounded Streams," *Proc. IEEE Int'l Conf. Data Eng. (ICDE),* 2003.

[21] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load-Shedding in a Data Stream Manager," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2003.

[22] N. Alon, Y. Matias, and M. Szegedy, "The Space Complexity of Approximating the Frequency Moments," *Proc. Ann. ACM Symp. Theory of Computing (STOC),* 1996.

[23] S. Guha, N. Koudas, and K. Shim, "Data-Streams and Histograms," *Proc. Symp. Theory of Computing,* 2001.

[24] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing Complex Aggregate Queries over Data Streams," *Proc. ACM SIGMOD,* 2002.

[25] S. Chandrasekaran and M.J. Franklin, "Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2004.

[26] S. Chaudhuri, R. Motwani, and V. Narasayya, "On Random Sampling over Joins," *Proc. ACM SIGMOD,* 1999.

[27] W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS," *Distributed and Parallel Databases,* vol. 1, no. 1, 1993.

[28] T. Urhan and M. Franklin, "XJOIN: A Reactively-Scheduled Pipelined Join Operator," *IEEE Data Eng. Bull.,* vol. 23, no. 2, 2000.

[29] J.-P. Dittrich, B. Seeger, D. Taylor, and P. Widmayer, "Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2002.

[30] Y. Tao, M.L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis, "RPJ: Producing Fast Join Results on Streams through Rate-Based Optimization," *Proc. ACM SIGMOD,* 2005.

[31] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M.A. Shah, "TelegraphCQ: An Architectural Status Report," *IEEE Data Eng. Bull.,* vol. 26, no. 1, 2003.

[32] W. Labio and H. Garcia-Molina, "Efficient Snapshot Differential Algorithms for Data Warehousing," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 1996.

[33] L. Wilburt, J. Wiener, H. Garcia-Molina, and V. Gorelik, "Efficient Resumption of Interrupted Warehouse Loads," *Proc. ACM SIGMOD,* 2000.

[34] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom, "Performance Issues in Incremental Warehouse Maintenance," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2000.

[35] A. Gupta and I.S. Mumick, "Maintenance of Materialized Views: Problems, Techniques, and Applications," *IEEE Data Eng. Bull.,* vol. 18, no. 2, 1995.

[36] H. Gupta and I. Mumick, "Incremental Maintenance of Aggregate and Outerjoin Expressions," to be published in *Information Systems.*

[37] X. Zhang and E.A. Rundensteiner, "Integrating the Maintenance and Synchronization of Data Warehouses Using a Cooperative Framework," *Information Systems,* vol. 27, no. 4, 2002.

[38] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View Maintenance in a Warehousing Environment," *Proc. ACM SIGMOD,* 1995.

[39] S. Chen, B. Liu, and E.A. Rundensteiner, "Multiversion-Based View Maintenance over Distributed Data Sources," *ACM Trans. Database Systems,* vol. 29, no. 4, pp. 675-709, 2004.

Neoklis Polyzotis received the PhD degree from the University of Wisconsin, Madison, in 2003. He is currently an assistant professor at the University of California, Santa Cruz. His research focuses on database systems and, in particular, on approximate query answering, online database tuning, and P2P databases. He is a recipient of a US National Science Foundation Faculty Early Career Development (CAREER) Award in 2004 and of an IBM Faculty Award in 2005 and 2006. He is a member of the IEEE.

Spiros Skiadopoulos received the diploma and the PhD degree from the National Technical University of Athens and the MPhil degree from UMIST. He is currently an assistant professor at the Department of Computer Science and Technology, University of Peloponnese. His research focuses on constraint databases and reasoning, query evaluation and optimization, and data warehouses. More information is available at www.uop.gr/~spiros.

Panos Vassiliadis received the PhD degree from the National Technical University of Athens in 2000. Since 2002, he has been with the Department of Computer Science, University of Ioannina, Greece, where he is also a member of the Distributed Management of Data (DMOD) Laboratory (http://www.dmod.cs.uoi.gr). He has published more than 50 papers in refereed journals and international conference proceedings on data warehousing, Web services, and database evolution, as well as a book on the fundamentals of data warehouses. He is a member of the ACM, the IEEE, and the IEEE Computer Society. More information is available at http://www.cs.uoi.gr/~pvassil.

Alkis Simitsis received the diploma and the PhD degree from the School of Electrical and Computer Engineering, National Technical University of Athens (NTUA) in 2000 and 2004, respectively. He is currently with the Computer Science Group, IBM's Almaden Research Center. Following that, he worked as a researcher in the Knowledge and Database Systems Lab, NTUA and as a visiting lecturer at the University of Peloponnese, Greece. His research interests include extraction-transformation-loading (ETL) processes in data warehouses, modeling and designing of data warehouses, query processing/optimization, keyword search, and data cleaning. He has published more than 30 papers in refereed journals and international conference proceedings in the above areas. More information is available at http://www.dblab.ntua.gr/~asimi.

Nils-Erik Frantzell is a computer science graduate from the University of California, Santa Cruz. He is currently working as a software design engineer in test at Microsoft. His interests include database management and next-generation file systems, psychology, and electronic music.