Supporting Streaming Updates in an Active Data Warehouse

Neoklis Polyzotis
Univ. of California - Santa Cruz
alkis@cs.ucsc.edu

Spiros Skiadopoulos
Univ. of Peloponnese
spiros@uop.gr

Panos Vassiliadis
Univ. of Ioannina
pvassil@cs.uoi.gr

Alkis Simitsis
Nat'l Tech. Univ. of Athens
asimi@dblab.ece.ntua.gr

Nils-Erik Frantzell
Univ. of California - Santa Cruz
nfrantze@ucsc.edu

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 on-line and thus achieves a higher consistency between the stored information and the latest data updates. The need for on-line 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*), that compensates for the difference in the access cost of the two join inputs by (a) relying entirely on fast sequential scans of R, and (b) 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.*

1. Introduction

Data warehouses are typically refreshed in a batch (or, off-line) 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 novel concept of *active* (or real-time) *data warehouses* [3, 13, 17, 22]. In this scenario, all updates to the production systems are propagated immediately to the warehouse and incorporated in an on-line 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. Figure 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.

1-4244-0803-2/07/$20.00 ©2007 IEEE.

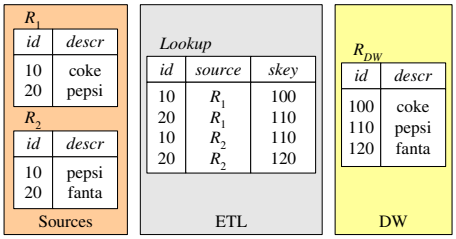


Figure 1. Surrogate key generation

The previous example is characteristic of several common transformations that take place in an active ETL process, such as duplicate detection or identification of newly inserted tuples. 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 on-line, 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 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). Our thesis is that the combination of these two elements, namely, the mismatch in input access speeds and the limited memory, forms a fundamentally different context compared to conventional warehousing architectures. As a result, join algorithms that are the norm in a conventional ETL process, such as hash join or sort-merge join, cannot provide effective support for active data warehousing.

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, the technical contributions of this paper 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: *(a)* it accesses R solely through fast sequential scans, and *(b)* it amortizes the cost of I/O operations over a large number of stream tuples. As we show in this paper, 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.

Experimental study of the performance 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 that MESHJOIN can accommodate update rates of up to 20,000 tuples/second under modest allocations of memory, outperforming by a factor of 10 a conventional join algorithm. Moreover, our study validates the accuracy of the analytical model in predicting the performance of the algorithm relative to its operating parameters.

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. We present our experimental study in Section 4 and cover related work in Section 5. We conclude the paper in Section 6.

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. We make no assumptions on the physical characteristics of R,

e.g., the existence of indices or its clustering properties, except that it is too large to fit in main memory. Since our focus is active warehousing, we assume that the warehouse receives S from the operational data sources in an on-line fashion. Thus, we henceforth model S as a streaming input, and use λ to denote the (potentially variable) arrival rate of update tuples. 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 operator. Combined with the (potentially) high arrival rate of S, it becomes obvious that the operator 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 operator 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 memory M, on the other hand, relates the performance of the operator 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 (a) 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, (b) the characterization of the algorithm's performance in terms of the service rate and the required memory resources.

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 set up 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: *(a)* it may require the (potentially expensive) maintenance of an additional index on R, and most importantly *(b)* 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. It seems necessary therefore to explore a new join algorithm that can specifically take into account the unique characteristics of the $S \bowtie R$ operation.

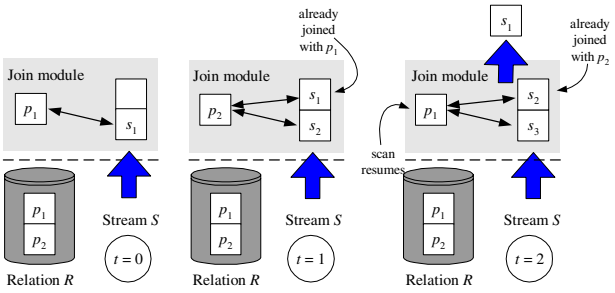


Figure 2. Operation of MESHJOIN

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. Assume that R contains 2 pages (p_1 and p_2) and that the join algorithm has enough memory to store a window of the 2 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 Figure 2.

– 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. Note that, at this point, page p_2 is joined with two stream tuples; 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 and brings in page p_1, and simultaneously replaces tuple s_1 with the next

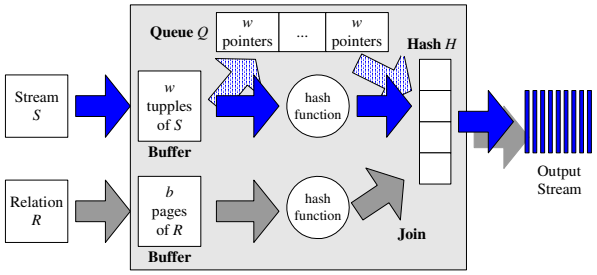


Figure 3. Data structures and architecture of MESHJOIN

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 the whole relation.

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. In more detail, 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). Figure 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.

Figure 4 shows the pseudo-code 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; its proof appears in the full version of the paper [18].

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

Figure 4. Algorithm MESHJOIN

Theorem 3.1 *Algorithm* MESHJOIN *correctly computes the exact join between a stream and a relation provided that* $\lambda \leq \mu$.

Hence, a basic assumption is that the join operator is fast enough to keep up with the arrival rate of update tuples. We examine this point 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 and Tuning

In this section, we develop a cost model for MESHJOIN and use it to tune the algorithm for different performance objectives.

Cost Model. Our cost model provides the necessary analytical tools to interrelate the following key parameters of the problem: *(a)* the stream rate λ, *(b)* the service rate μ of the join, and *(c)* 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. For ease of reference, 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: *(a)* the buffer of R uses $b \cdot v_P$ bytes, *(b)* the buffer of S uses $w \cdot v_S$ bytes, *(c)* 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 *(d)* 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:

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

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

Table 1. Notation of the cost model

Having related w and b to the required memory, we now shift our attention to the processing speed of the algorithm. 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:

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

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

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

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 Indexed Nested Loops 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 Theorem 3.1 and Equation 3, 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} (Equation 2), we arrive at an inequality that links the parameters of MESHJOIN (namely, w and b) to the arrival rate of the stream. Combined with Equation 1 that links w and b to the memory requirements of the operator, the previous expression forms our basic tool for tuning MESHJOIN according to different objectives.

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 micro-benchmarks.

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 linearly depends on w (Equation 1). Therefore, to minimize M, we have to minimize w.

2. The minimum value for w is specified by Equation 5 as follows:
$$w = \lambda c_{loop} \tag{6}$$
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 Equation 1, thus specifying M as a function of b. Using standard calculus methodology, we can find exactly the value of b that minimizes M (more details can be found in the long version of the paper [18]). Given Equation 1, this also implies that

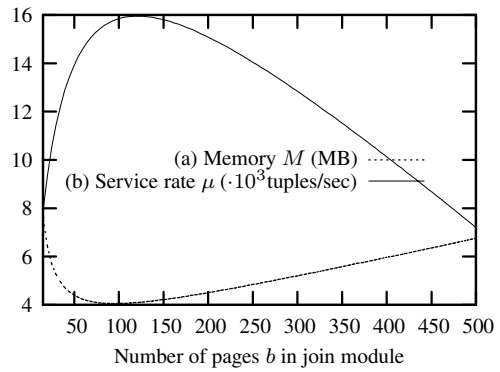


Figure 5. (a) Minimizing M (μ is fixed) and (b) Maximizing μ (M is fixed)

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 Figure 5(a), that shows M as a function of b for typical values of the cost factors. As shown, 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.

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 (Equations 1, 2 and 3 respectively), we can specify μ as a function of b, and subsequently find the value that maximizes μ with standard calculus methodology.

Figure 5(b) shows the relationship between μ and b for sample values of the cost factors. 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 (Equation 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.

3.3. Extensions

In this section, we discuss possible extensions of the basic MESHJOIN scheme that we introduced previously. In the interest of space, we provide only a short overview and defer the details to the full version of the paper [18].

Approximate join processing. MESHJOIN can handle a stream rate that is too high for the available memory, by selectively dropping S-tuples and thus generating an approximation of the result. The shedding strategy affects directly the properties of the approximation, and thus must be chosen carefully to match the semantics of the underlying transformation.

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. For all practical purposes, this situation does not compromise the correctness of the ETL transformations that we consider in this paper. 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 same as detailed in Figure 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 speed-up 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.

Dynamic tuning. Up to this point, we have assumed that parameters w and b remain fixed for the operation of MESHJOIN. While the algorithm can readily handle a change of w in mid-flight, it is possible to further extend MESHJOIN so that it can accommodate a varying b without compromising the correctness of the generated results. In the interest of space, the details can be found in the full version of this paper [18]. The end goal is to extend MESHJOIN with a self-tuning mechanism that 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.

(This mechanism would rely of course on the analytical cost model described previously.) We plan to explore this topic as part of our future work on MESHJOIN.

4. Experiments

In this section, we present an experimental study that we have conducted in order to evaluate 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.

4.1. 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. As mentioned from the beginning, we abstract a large set of ETL processes as the join between a stream of updates and a disk based relation. We consider two join processing techniques in our experiments.

– MESHJOIN. We have completed a prototype implementation of the MESHJOIN algorithm that we introduce in this paper. 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.

– *Index-Nested-Loops.* We have implemented a join module based on the Indexed Nested Loops (INL) 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.

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

– *Synthetic Data Set.* Table 2 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% 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.

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

Table 2. Data Set Characteristics

– *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 [11]. We use measurements from two different months to create a relation and a stream of update tuples, both consisting of 10 million tuples each. The tuple-size is 32 bytes for both R and S and the underlying value domain is $[0, 36000]$.

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.

Experimental Platform. We have performed our experiments on a Pentium IV 3GHz machine with 1GB 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. (We note that the MESHJOIN algorithm is less sensitive to buffering as it performs continuous sequential scans over the large disk-based relation.)

4.2. 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 21MB (5% of the relation size) and we vary (b, 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.

Figure 6(a) 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 21MB.) 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. 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% (M=200KB) up to 10% (M=40MB). All reported measurements are with a cold cache.

Figure 6(b) 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 4MB (1% 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/sec 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.

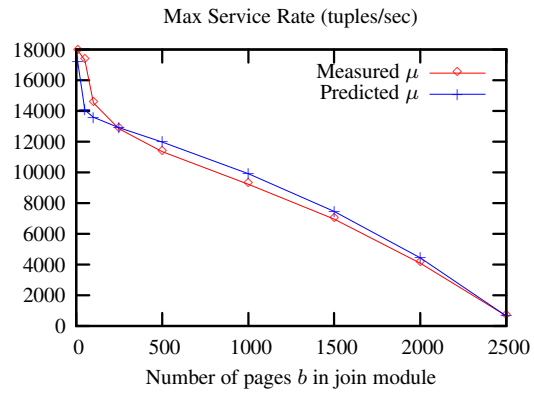
Compared to INL, MESHJOIN is the clear winner as it achieves a 10x improvement for all memory allocations. For an allotted memory M of 2MB (0.5% of the total relation size), for instance, INL can sustain 274 tuples/second while MESHJOIN achieves a service rate of 3500 tuples/second. In essence, the buffer 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.

We have also performed experiments with different skews in the join values, in both the relation and the stream. Our results have been qualitatively the same and are omitted in the interest of space.

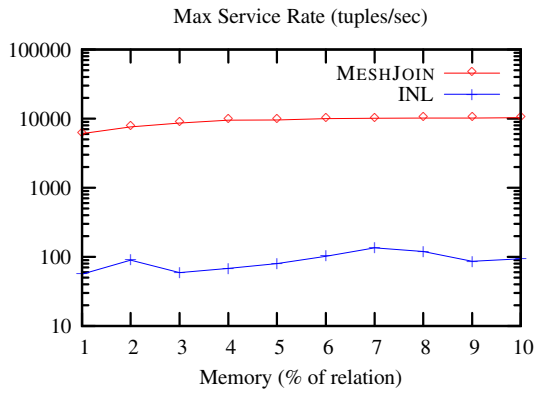
Varying skew. In the second set of experiments, we measure the performance of MESHJOIN for different values of the relation skew parameter z_R. Recall that z_R controls the distribution of values in the join column of R and hence affects the selectivity of the join. We keep the skew of the stream fixed 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 20MB of main memory (5% of the size of R).

Figure 6(c) 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, with a minimum rate of 9,000 tuples/sec for the highest skew. Compared to INL, it offers significant improvement in all cases and is again the clear winner.

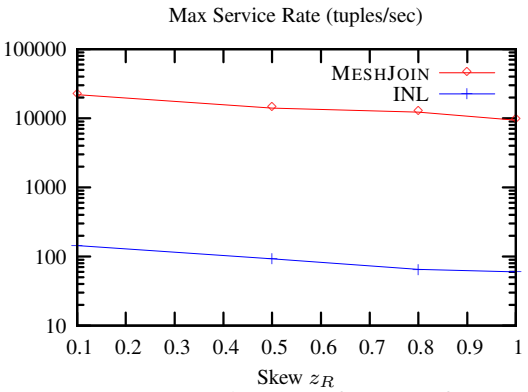
We note that we have also performed experiments by varying the skew z_S of the stream. Our results have shown that both techniques are relatively insensitive to this parameter and are thus omitted in the interest of space.



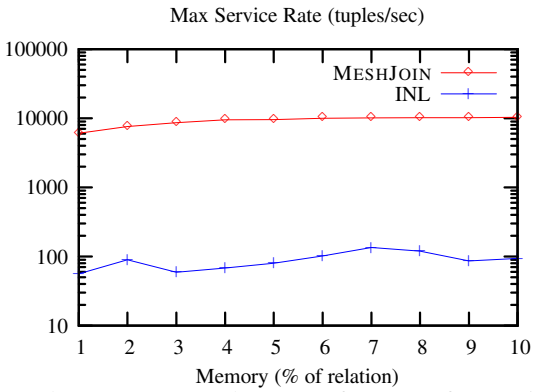
(a) MESHJOIN: predicted and measured performance (synthetic data)



(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)

Figure 6. Experimental evaluation of MESHJOIN.

Performance of MESHJOIN on real-life data sets. As a final experiment, 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% (4MB) to 10% (40MB). Again, we compare MESHJOIN to INL, using the service rate as the evaluation metric.

Figure 6(d) 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.

5. Related Work

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 [8].)

Active or real-time data warehousing has recently appeared in the industrial literature [3, 17, 22]. 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 [14, 15, 16]. Contrary to our setting, these algorithms are designed to operate in a batch, off-line fashion. Work in materialized views refreshment [9, 10, 23, 24] is also relevant, but orthogonal to our setting. The crucial decision concerns whether a view can be updated given a delta set of updates.

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 infinite relations [1, 2, 6, 20]. Earlier studies [4, 7, 12, 21] have introduced generalizations of Symmetric Hash-Join to a multi-way join operator in order to efficiently handle join queries over multiple unbounded streams. These works, however, 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. Works for the join of streamed bounded relations, like the Progressive Merge Join [5], and the more recent Rate-based Progressive Join [19] propose join algorithms that access the streaming inputs continuously and maintain the received tuples in memory in order to generate results as early as possible; when the received input exceeds the capacity of main-memory, the algorithm flushes a subset of the data to disk and processes it later when (CPU or memory) resources allow it. Clearly, this model does not match well the constraints of our setting, since the buffering of S tuples would essentially stall the stream of updates and thus compromise the requirement for on-line refreshing.

6. Conclusions

In this paper, 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 limited main memory.

Based on the above results, research can be pursued in different directions. Most importantly, multi-way joins between a stream and many relations is a research topic that requires the fine-tuning of the iteration of the multiple relations in main memory as the stream tuples flow through the join operator(s). The investigation of other common operators for active warehousing (e.g., multiple on-line aggregation) is another topic for future work.

Acknowledgements. This work supported by PYTHAGORAS EPEAEK II programme, EU and Greek Ministry of Education, co-funded by the European Social Fund (75%) and National Resources (25%).

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of PODS*, 2002.

[2] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), 2001.

[3] D. Burleson. New developments in oracle data warehousing. *Burleson Consulting*, April 2004.

[4] S. Chandrasekaran and M. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB J.*, 12(2), 2003.

[5] J.-P. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. Progressive merge join: A generic and non-blocking sort-based join algorithm. In *Proc. of VLDB*, 2002.

[6] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2), 2003.

[7] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of VLDB*, 2003.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2), 1993.

[9] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2), 1995.

[10] H. Gupta and I. Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Inf. Syst.*, to appear.

[11] C. J. Hahn, S. G. Warren, and J. London. Edited synoptic cloud reports from ships and land stations over the globe, 1982–1991.

[12] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of VLDB*, 2003.

[13] A. Karakasidis, P. Vassiliadis, and E. Pitoura. ETL queues for active data warehousing. In *Proc. of IQIS*, 2005.

[14] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proc. of VLDB*, 1996.

[15] W. Labio, J. L. Wiener, H. Garcia-Molina, and V. Gorelik. Efficient resumption of interrupted warehouse loads. In *Proc. of SIGMOD*, 2000.

[16] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. In *Proc. of VLDB*, 2000.

[17] Oracle Corp. On-time data warehousing with oracle10g - information at the speed of your business. *An Oracle White Paper*, August 2003.

[18] A. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell. Supporting Streaming Updates in an Active Data Warehouse. Technical report, University of California Santa Cruz, 2006.

[19] Y. Tao, M. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. RPJ: Producing fast join results on streams through rate-based optimization. In *Proc. of SIGMOD*, 2005.

[20] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc. of SIGMOD*, 1992.

[21] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proc. of VLDB*, 2003.

[22] C. White. Intelligent business strategies: Real-time data warehousing heats up. *DM Review*, 2002.

[23] X. Zhang and E. A. Rundensteiner. Integrating the maintenance and synchronization of data warehouses using a cooperative framework. *Inf. Syst.*, 27(4), 2002.

[24] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. of SIGMOD*, 1995.