Meshing Streaming Updates with Persistent Data in an Active Data Warehouse

Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, 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 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 article, 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.

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

I. 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 concept of active (or real-time) data warehouses [1]–[4]. 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. 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.

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 $S \bowtie C R$ as a core operation, where $S$ is the relation of source updates, $R$ is a large, disk-resident, warehouse relation, and the join condition $C$ 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 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). 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

Fig. 1. Surrogate key generation

<table>
<thead>
<tr>
<th>$R_1$ id</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>coke</td>
</tr>
<tr>
<td>20</td>
<td>pepsi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_2$ id</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>pepsi</td>
</tr>
<tr>
<td>20</td>
<td>fanta</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lookup id</th>
<th>source</th>
<th>skey</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$R_1$</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>$R_1$</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>$R_2$</td>
<td>110</td>
</tr>
<tr>
<td>20</td>
<td>$R_2$</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R_{up}$ id</th>
<th>descr</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>coke</td>
</tr>
<tr>
<td>110</td>
<td>pepsi</td>
</tr>
<tr>
<td>120</td>
<td>fanta</td>
</tr>
</tbody>
</table>

Sources ETL DW

N. Polyzotis is with the Univ. of California - Santa Cruz.
S. Skiadopoulos is with the Univ. of Peloponnese.
P. Vassiliadis is with the Univ. of Ioannina.
A. Simitsis is with the Nat’l Tech. Univ. of Athens.
N.-E. Frantzell is with the Univ. of California - Santa Cruz.
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 article 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 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. This leads of course to an approximation of the true join result due to the loss of some answer tuples. 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 off-line strategy that has a-priori knowledge of the complete update stream. While the latter is clearly not viable in practice, it provides a good benchmark for the performance of any on-line strategy.

- **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 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. Finally, we show that the TOPW heuristic yields similar performance to the optimal off-line strategy under a wide-range of settings, and is thus an effective shedding strategy when memory is under-provisioned.

The remainder of the article is structured as follows. In Section II, we define the problem more precisely and discuss the requirements for an effective solution. Section III provides a detailed definition of the proposed algorithm, including its analytical cost model and its tuning for different objectives. Section IV examines the problem of approximate join processing and discusses different shedding strategies. We present our experimental study in Section V and cover related work in Section VI. We conclude the article in Section VII.

II. 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 \( S \bowtie C \times R \), where \( S \) is the relation of source updates, \( R \) is a large relation stored in the data staging area, and \( C \) depends on the transformation. To simplify our presentation, we henceforth assume that \( C \) 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 \( \lambda \) 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 \( M_{\text{max}} \) 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 \( \mu \), and the consumed memory \( M \). The service rate \( \mu \) 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 operator to the resources that it requires. (We assume that \( M \leq M_{\text{max}} \).) 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 article 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 at 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.

III. 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 without other assumptions for the stream or the relation, and the required memory resources.

A. Algorithm Definition

Before describing the MESHJOIN algorithm in detail, we illustrate its key idea using a simplified example. Assume that \( R \) contains 2 pages \((p_1\text{ and } p_2)\) 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 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. 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 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. 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.

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

**Proof:** We need to ensure that the MESHJOIN algorithm is exact (there is no stream tuple loss) and correct (i.e., stream tuples are joined with all tuples of the relation exactly one time). Let \( \beta(1) \) (respectively \( \beta(2) \), ..., \( \beta(N_b) \)) be a set that contains the first (respectively second, ..., last) \( b \) pages of relation \( R \). Let also \( \omega(1) \) (respectively \( \omega(2) \), ...) be a set that contains the first (respectively second, ..., \( b \)) tuples of stream \( S \).

We will follow the execution of MESHJOIN algorithm step by step. We will use \( k \) as a counter of the **While** loop of the algorithm. The next table summarizes the first loop of MESHJOIN algorithm \((k = 1)\).

<table>
<thead>
<tr>
<th>1st loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>The algorithm reads ( \beta(1) ) and ( \omega(1) ).</td>
</tr>
<tr>
<td>Inserts ( \omega(1) ) into hash ( H ).</td>
</tr>
<tr>
<td>Inserts ( w ) pointers (referencing to the ( \omega(1) ) tuples of ( H )) into queue ( Q ).</td>
</tr>
<tr>
<td>Joins ( \beta(1) ) with the tuples of the hash ( H ).</td>
</tr>
<tr>
<td>Outputs ( \beta(1) \bowtie \omega(1) ).</td>
</tr>
</tbody>
</table>

During the first loop, the algorithm has computed \( \beta(1) \bowtie \omega(1) \).

Since queue \( Q \) accommodates \( \lfloor N_b \rfloor \) pointers, it follows that for every \( k \) loop, where \( 1 \leq k \leq \lfloor N_b \rfloor \), there is room in \( Q \) to add the incoming \( w \) tuples of \( S \). Thus, we have:

<table>
<thead>
<tr>
<th>( k )th loop, ( 1 \leq k \leq \lfloor N_b \rfloor )</th>
</tr>
</thead>
<tbody>
<tr>
<td>The algorithm reads ( \beta(k) ) and ( \omega(k) ).</td>
</tr>
<tr>
<td>Inserts ( \omega(k) ) into hash ( H ).</td>
</tr>
<tr>
<td>Inserts ( w ) pointers (referencing to the ( \omega(k) ) tuples of ( H )) into queue ( Q ).</td>
</tr>
<tr>
<td>Joins ( \beta(k) ) with the tuples of the hash ( H ).</td>
</tr>
<tr>
<td>Outputs ( \beta(k) \bowtie (\omega(1) \cup \cdots \cup \omega(k)) ).</td>
</tr>
</tbody>
</table>

The data structures of MESHJOIN, during the above loops, are depicted in Fig. 5(a). Up to the \( k \)th loop, the algorithm has computed:

\[
\beta(1) \bowtie (\omega(1) \cup \cdots \cup \omega(k)) \bowtie (\omega(1) \cup \cdots \cup \omega(\lfloor N_b \rfloor))
\]

Thus, at the \( \lfloor N_b \rfloor \)th loop, MESHJOIN algorithm computes:

\[
\beta(1) \bowtie (\omega(1) \cup \cdots \cup (\omega(1) \cup \cdots \cup (\omega(\lfloor N_b \rfloor) \bowtie (\omega(\lfloor N_b \rfloor) \bowtie (\omega(\lfloor N_b \rfloor)))))
\]

or equivalently:

\[
\omega(1) \bowtie (\beta(1) \cup \cdots \cup \beta(\lfloor N_b \rfloor)) \cup \cdots \cup \omega(\lfloor N_b \rfloor) \bowtie (\beta(\lfloor N_b \rfloor))
\]

Since \( R = \beta(1) \cup \cdots \cup \beta(\lfloor N_b \rfloor) \), we also have:

\[
\omega(1) \bowtie R \cup \omega(2) \bowtie (R - \{\beta(1)\}) \cup \cdots \cup \omega(\lfloor N_b \rfloor) \bowtie (\beta(\lfloor N_b \rfloor))
\]

In other words, set \( \omega(1) \) has been joined with all the pages of relation \( R \) and can thus be removed from the system. This is done in the \( \lfloor N_b \rfloor + 1 \)th loop of the MESHJOIN algorithm. Notice that the pointers in queue \( Q \) that reference to the tuples of \( \omega(1) \) are in the head of \( Q \). Thus, the \( \lfloor N_b \rfloor + 1 \)th loop just dequeues \( w \) pointers and deletes the corresponding tuples of the hash \( H \).

Specifically, the \( k = \lfloor N_b \rfloor + 1 \)th loop of the MESHJOIN algorithm performs the following operations:

<table>
<thead>
<tr>
<th>( \lfloor N_b \rfloor + 1 )th loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>The algorithm reads ( \beta(1) ) and ( \omega(\lfloor N_b \rfloor + 1) ).</td>
</tr>
<tr>
<td>Inserts ( \omega(\lfloor N_b \rfloor + 1) ) into hash ( H ).</td>
</tr>
<tr>
<td>Dequeues ( w ) pointers from ( Q ) (referencing to ( \omega(1) )) and deletes the corresponding tuples of ( H ).</td>
</tr>
<tr>
<td>Inserts ( w ) pointers (referencing to the ( \omega(\lfloor N_b \rfloor + 1) ) tuples ( H )) into queue ( Q ).</td>
</tr>
<tr>
<td>Joins ( \beta(1) ) with the tuples of the hash ( H ).</td>
</tr>
<tr>
<td>Outputs ( \beta(1) \bowtie (\omega(2) \cup \cdots \cup \omega(\lfloor N_b \rfloor + 1)) ).</td>
</tr>
</tbody>
</table>

Generalizing, for \( \lfloor N_b \rfloor + 1 \leq k \), the MESHJOIN algorithm carries out the following operations:

<table>
<thead>
<tr>
<th>( k )th loop, ( \lfloor N_b \rfloor + 1 \leq k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>The algorithm reads ( \beta(k - \lfloor N_b \rfloor) ) and ( \omega(k) ).</td>
</tr>
<tr>
<td>Inserts ( \omega(k) ) into hash ( H ).</td>
</tr>
<tr>
<td>Dequeues ( w ) pointers from ( Q ) (referencing to ( \omega(k - \lfloor N_b \rfloor) )) and deletes the corresponding tuples of ( H ).</td>
</tr>
<tr>
<td>Inserts ( w ) pointers (referencing to the ( \omega(k) ) tuples ( H )) into queue ( Q ).</td>
</tr>
<tr>
<td>Joins ( \beta(k - \lfloor N_b \rfloor) ) with the tuples of the hash ( H ).</td>
</tr>
<tr>
<td>Outputs ( \beta(k - \lfloor N_b \rfloor) \bowtie (\omega(k - \lfloor N_b \rfloor + 1) \cup \cdots \cup \omega(k)) ).</td>
</tr>
</tbody>
</table>

The data structures of MESHJOIN, during the above loops, are depicted in Fig. 5(b). Up to the \( k \)th loop \((\lfloor N_b \rfloor + 1 \leq k)\), the algorithm has computed:
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: (a) the stream rate $\lambda$, (b) the service rate $\mu$ 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. The eventual goal is to employ this cost model in order to tune the parameters of the algorithm based on the characteristics of the input. For ease of reference, the notation used in our discussion is summarized in Table I.

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 $w \cdot \frac{N_b}{b} \cdot \text{sizeof} (\text{ptr})$ bytes (where sizeof(ptr) is the size of a pointer) and (d) the hash table $H$ uses $w \cdot f \cdot \frac{N_v}{b} \cdot v_S$ bytes (where $f$ is the fudge factor of the hash table implementation). Thus, we have:

$$M = b \cdot v_p + w \cdot v_S + w \cdot \frac{N_b}{b} \cdot \text{sizeof}(\text{ptr}) + w \cdot f \cdot \frac{N_v}{b} \cdot v_S \leq M_{\text{max}}$$  \hspace{1cm} (2)

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 $c_{\text{loop}}$ 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 I and are straightforward to measure in an actual implementation of MESHJOIN. In total, we can express the cost $c_{\text{loop}}$ as follows:

$$c_{\text{loop}} = c_{I/O}(b) + w \cdot c_E + (\text{Read } b \text{ pages})$$
$$w \cdot c_S + (\text{Read } w \text{ tuples from } Q \text{ and } H)$$
$$w \cdot c_A + (\text{Read } w \text{ tuples from the stream buffer})$$
$$b \cdot v_p \cdot c_H + (\text{Add } w \text{ tuples to } Q \text{ and } H)$$
$$\sigma \cdot v_b \cdot c_O \quad (\text{Probe } H \text{ with } R\text{-tuples}) \quad (\text{Construct results})$$

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

$$\mu = \frac{w}{c_{\text{loop}}} \quad (4)$$

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:

$$IO_s = \frac{b}{w} \quad \text{and} \quad IO_t = \frac{b}{c_{\text{loop}}} \quad (5)$$

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 4, we can derive the relation between $\lambda$, $c_{\text{loop}}$, and $w$.

$$\lambda \leq \mu \Rightarrow \lambda c_{\text{loop}} \leq w \quad (6)$$

By substituting the expression for $c_{\text{loop}}$ (Equation 3), we arrive at an inequality that links $w$ and $b$ to the arrival rate of the stream. Combined with Equation 2 that links $w$ and $b$ to the memory requirements of the operator, the previous expression forms our
C. 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 \( \mu \), and maximizing the service rate \( \mu \) 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 I, 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.

In what follows, we discuss the details of the tuning methodology. We first examine the off-line 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 on-line tuning, where the algorithm continuously monitors the characteristics of the stream and adapts on-the-fly the parameters of the join.

**Off-line Tuning.** As mentioned earlier, we consider two objectives when tuning MESHJOIN: minimizing memory consumption, and maximizing the service rate. We consider them in this order below.

**Minimizing \( M \).** In this case, we assume that the stream rate \( \lambda \) 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 2). Therefore, to minimize \( M \), we have to minimize \( w \).

2) The minimum value for \( w \) is specified by Equation 6 as follows:

\[
    w = \lambda c_{\text{loop}} \quad (7)
\]

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 2, thus specifying \( M \) as a function of \( b \). Using standard calculus methodology, we can find exactly the value of \( b \) that minimizes \( M \). Given Equation 2, 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. 6(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 \( \mu \).** Here, we assume that the available memory for the algorithm \( M \) is fixed, and we are interested in maximizing the service rate \( \mu \). Using the expressions for \( M \), \( c_{\text{loop}} \) and \( \mu \) (Equations 2, 3 and 4 respectively), we can specify \( \mu \) as a function of \( b \), and subsequently find the value that maximizes \( \mu \) using standard calculus methodology.

Fig. 6(b) shows the relationship between \( \mu \) and \( b \) for sample values of the cost factors. We observe that \( \mu \) 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 5). 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.

**On-line 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 on-line 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 mid-flight change of \( b \). The original algorithm can readily handle a mid-flight 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 pseudo-code for AMESHJOIN is shown in Fig. 7. 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 $\in [0, NR]$. 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 $[\text{currPage}, \text{currPage} + b) \% NR$ 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 3 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 $(\text{ptr}(s_0), 0)$ to queue Q (where $\text{ptr}(s_0)$ 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:

<table>
<thead>
<tr>
<th>currPage</th>
<th>Joined with pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(\text{ptr}(s_0), 0)$ $p_0$</td>
</tr>
</tbody>
</table>

- **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 $(\text{ptr}(s_1), 1)$ 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:

<table>
<thead>
<tr>
<th>currPage</th>
<th>Joined with pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$(\text{ptr}(s_0), 0)$ $p_0\langle p_1$</td>
</tr>
<tr>
<td></td>
<td>$(\text{ptr}(s_1), 1)$ $p_1$</td>
</tr>
</tbody>
</table>

- 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 $[\text{currPage}, \text{currPage} + b) \% NR = [2, 4) \% NR = [2, 3) \cup [0, 1]$ in line 4). Then, the algorithm places $s_2$ to hash H and $(\text{ptr}(s_2), 2)$ 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 $s_0$ and $s_1$, and sets currPage = 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:

<table>
<thead>
<tr>
<th>currPage</th>
<th>Joined with</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(\text{ptr}(s_1), 1)$ $p_1, p_2$</td>
</tr>
<tr>
<td></td>
<td>$(\text{ptr}(s_2), 2)$ $p_2$</td>
</tr>
</tbody>
</table>

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:

**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 $[\text{currPage}, \text{currPage} + b) \% NR$ from R
5. Add the w tuples of S in H
6. **ENQUEUE** in Q, $(w, \text{currPage})$ where w are pointers to the above tuples in H
7. **For** i = 1 to b
8. Join in-memory page currPage with H
9. If (currPage = head(Q).startPage – 1) \% NR
10. **DEQUEUE** head(Q)
11. Remove the tuples in H that correspond to head(Q)
12. **EndIf**
13. currPage $\leftarrow$ (currPage + 1) \% NR
14. **EndFor**
15. **EndWhile**

**Fig. 7. Algorithm AMESHJOIN**

The following theorem formalizes the correctness of the algorithm.

**Theorem 3.2:** AMESHJOIN computes correctly $S \bowtie R$.

**Proof:** We show that the following property holds when head(Q) is dequeued: the stream tuples in head(Q) have been joined exactly once with all the pages in R. This property will guarantee the correctness of the algorithm. The proof works by induction on the arrival order k of the packet in head(Q).

**Base case:** $k=1$. This applies to the first packet that is enqueued after the algorithm starts. Thus, head(Q).startPage = 0. Clearly, the algorithm will join every page in [0, NR) with head(Q) before dequeuing the packet.

**Inductive step.** Let w be the $k+1$-th packet that is enqueued, and let $w'$ be the k-th packet. If $w'$ is dequeued before w is enqueued, then w is the only packet in Q and the claim is proven similar to the base case. When w is enqueued, $w'$ has already been joined with the pages in the range $[w'.startPage, currPage)\% NR = [w'.startPage, w.startPage)\% NR$. From that point onward, the two packets are joined with the same pages. After page currPage = $(w'.startPage – 1) \mod NR$, then $w' \equiv$ (head(Q) is dequeued correctly and w comes at the top of Q. Since $w'$ is joined exactly once with all pages in R, $w'$ has been joined exactly once with all R except for pages in the range $[w'.startPage, w.startPage)\% NR$. The succeeding iterations will join w with the pages in the range $[w'.startPage, w.startPage)\% NR$. Thus, w will have joined exactly once with every page in R when it is dequeued.

Using the AMESHJOIN algorithm, we can devise 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 III-B 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
D. 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 \( \tau_s \) and \( \tau'_s \) that have the same join key and enter the sliding window in the same batch. Assuming that \( \tau_1, \tau_2, \ldots \) are the joining \( R \)-tuples for \( \tau_s \) and \( \tau'_s \), it is straightforward to verify that the output stream will have the form \( (\tau_s, r_1) (\tau'_s, r_1) (\tau_s, r_2) (\tau'_s, r_2) \ldots \) whereas an order-preserving output would group all the results of \( \tau_s \) and \( \tau'_s \) 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 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 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.

**IV. 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 (Figure 8(a)). 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: (a) the trend of the stock’s price over the last week and (b) 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 in 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 is 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% completeness and consistency of the incoming streaming data for more resources (I/O, CPU, memory) that can be shared among the other processes.

Our approach to approximate query processing is based on tuple-shedding, i.e., dropping a fraction of the incoming stream tuples and thus reducing the effective arrival rate of the stream. Fig. 9(b) provides an illustration. The data that is shed can be stored and propagated to the data warehouse in a later idle time. Thus, the ETL process has two variants: a lightweight one, which is used to provide end-users with as fresh data as possible and a regular one that synchronizes the stored data at the warehouse, in order to guarantee 100% completeness and consistency with respect to the sources. This can be achieved either off-line, or at periods with low user workload and possibly requires some extra synchronization with the sources (as traditionally happens in ETL processes.).

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. Yet 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.
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.

### A. Overview of Problem Space

Formally, let $\lambda$ be the current arrival rate of the stream and $\mu$ the current service rate of the algorithm. Based on the cost model of Section III-A, this service rate is determined by the cost $c_{\text{loop}}$ of a single iteration and the block size $w$ of stream tuples that is processed. Hence, a fast stream will deposit $c_{\text{loop}} \lambda > w$ new stream tuples at the end of each iteration, implying that the join algorithm will fall behind at a rate of $\lambda - w/c_{\text{loop}}$ tuples per second. The goal of the shedding mechanism, therefore, is to keep a fraction $w/(\lambda c_{\text{loop}})$ 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, on the other hand, 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, however, 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>
<thead>
<tr>
<th></th>
<th>MAX-SUBSET</th>
<th>RANDOM-SAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many-to-One</td>
<td>Keep $w$</td>
<td>Sample $w$</td>
</tr>
<tr>
<td>Many-to-Many</td>
<td>TOPW</td>
<td>Cluster Sampling [7]</td>
</tr>
</tbody>
</table>

**TABLE II**

**STREAM SHEDDING STRATEGIES.**

Table II 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 $\lambda c_{\text{loop}}$ 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 $\lambda c_{\text{loop}} - w$ result tuples.
- **Many-to-One/RANDOM-SAMPLE.** In this case, it suffices to maintain a random uniform sample of size $w$ out of the $\lambda c_{\text{loop}}$ 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 $w/(\lambda c_{\text{loop}})$ [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. In short, 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.

### B. Shedding Strategies for Many-to-Many/MAX-SUBSET

In what follows, we analyze shedding strategies suitable for the sub-space 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 on-line 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 off-line shedding strategy, termed OPTOFFLINE, 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 $\tau_i$. (The switch to finite streams is necessary so that an off-line strategy becomes meaningful. Clearly, the on-line TOPW heuristic can be applied on infinite streams.) For each stream tuple $\tau_i$ we will use $f_R(i)$ to denote the count of $R$-tuples that join with $\tau_i$. We assume that this information is readily available from the optimizer statistics that the database system maintains on $R$.

**On-line 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, of course, is to maximize the number of join results. More formally, let $t_r$ 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 \([t_r, t_r + c_{loop}]\) ordered by their frequency values \( f_R(i) \). Hence, MESHJOIN receives the \( w \) tuples from the interval \([t_r, t_r + c_{loop}]\) 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. Figure 10 illustrates this case with a sample stream of 10 elements. Assume that \( w = 2 \) and \( c_{loop} = 4 \) time units. For each stream element, the figure shows the corresponding frequency \( f_R(i) \), i.e., how 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 illustrates, 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.

Off-Line Optimal Shedding. We now present an off-line 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 on-line heuristic.

Similar to TOPW, OPTOFFLINESHED maintains a working buffer of \( w \) tuples that is consumed by MESHJOIN. As Figure 10 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: (a) the algorithm has to select tuples from \( s_i \) onward, (b) MESHJOIN will make its next request for tuples at time-stamp \( lt \), and (c) the algorithm has already selected \( sb \leq w \) tuples in the buffer. Hence, \( optloss(0,0,0) \) denotes the optimal loss for the whole stream. For completeness, we assume that \( optloss(i,lt,0) = 0 \) if \( i > |S| \).

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:

\[
\text{optloss}(i,lt,0) = \min(\text{optloss}(i+1,lt,0) + f_R(i))
\]

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 \( i \geq lt \), 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:

\[
\text{optloss}(i,lt,0) = \begin{cases} 
\text{optloss}(i+1,lt,0) + f_R(i) & i < lt \\
\text{optloss}(i,i+c_{loop},0) & i \geq lt 
\end{cases}
\]

The previous equations form the basis behind the dynamic programming algorithm for computing the optimal loss for the particular stream. The pseudo-code for the algorithm is shown in Figure 11. The algorithm iterates over all values of \( i \) and maintains two tables, namely, \( T[i,|S|+1,0,0, w] \) and \( T[i,|S|+1,0,0, w] \), that store the entries for \( optloss(i,lt,0,0, w) \) and \( optloss(i+1,lt,0,0, w) \) 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 \( optloss(0,0,0,0, w) \) can be retrieved from \( T^*[0,0,0,0, w] \) 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 \( optloss(0,0,0,0, w) \) quite efficiently, with a space complexity of \( O(|S|^2 w) \) 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 on-line shedding strategy in Many-to-Many/MAX-SUBSET that maintains a working buffer of \( w \) tuples and achieves an effective arrival rate of \( w/c_{loop} \).

V. 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.

A. 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 III-A). 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 III 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.

– 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]. 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 $[0,36000]$. Each input comprises 10 million tuples.

Evaluation Metrics. We evaluate the performance of a join algorithm based on its service rate $\mu$, 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.

B. 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 III-A. We use the synthetic data set with a fixed memory budget of 21MB (5% 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. 12(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.

Fig. 12(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
alocation 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 $R$).

Fig. 12(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 9,900 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.

**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% (4MB) to 10% (40MB). Again, we compare MESHJOIN to INL, using the service rate as the evaluation metric.

Fig. 12(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.

**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 IV), 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 use *TopW* only in our study as it represents a novel aspect of our work.

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 stream rate $\lambda$ and the service rate $\mu$ 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 $\rho$ and shedding algorithm, we measure the average fraction of join results over 10 different streams, each containing $\rho \times 20,000$ 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.

Fig. 13 depicts the fraction of generated join results for TopW as a function of the speed ratio $\rho$, 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 the results indicate, 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% 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, on the other hand, implies equal contributions from all stream tuples and hence bigger losses overall. The important observation, however, is that the *on-line* performance of TopW is comparable to the *off-line* performance of the optimal shedding algorithm. These results clearly verify the effectiveness of our TopW heuristic as a practical load-shedding mechanism for fast streams.

VI. 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 [10].)

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 [11]–[13]. Contrary to our setting, these algorithms are designed to operate in a batch, off-line fashion. Work in
materialized views refreshment [14]–[17] 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 [18]–[21]. Earlier studies [22]–[25] 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 [26], and the more recent Rate-based Progressive Join [27] 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.

VII. CONCLUSIONS

In this article, 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 has been supported by programs PYTHAGORAS II and PENED 03, co-funded by the European Union - European Social Fund (ESF) and National Sources.

REFERENCES