Semi-Streamed Index Join for Near-Real Time Execution of ETL Transformations

Mihaela A. Bornea #1, Antonios Deligiannakis *2, Yannis Kotidis #1, Vasilis Vassalos #1

#*Athens U. of Econ and Business, Technical University of Crete*

1{mihaela,kotidis,vassalos}@aueb.gr 2adeli@softnet.tuc.gr

***Abstract*—Active data warehouses have emerged as a new business intelligence paradigm where data in the integrated repository is refreshed in near real-time. This shift of practices achieves higher consistency between the stored information and the latest updates, which in turn influences crucially the output of decision making processes. In this paper we focus on the changes required in the implementation of Extract Transform Load (ETL) operations which now need to be executed in an online fashion. In particular, the ETL transformations frequently include the join between an incoming stream of updates and a disk-resident table of historical data or metadata. In this context we propose a novel Semi-Streaming Index Join (SSIJ) algorithm that maximizes the throughput of the join by buffering stream tuples and then judiciously selecting how to best amortize expensive disk seeks for blocks of the stored relation among a large number of stream tuples. The relation blocks required for joining with the stream are loaded from disk based on an optimal plan. In order to maximize the utilization of the available memory space for performing the join, our technique incorporates a simple but effective cache replacement policy for managing the retrieved blocks of the relation. Moreover, SSIJ is able to adapt to changing characteristics of the stream (i.e. arrival rate, data distribution) by dynamically adjusting the allocated memory between the cached relation blocks and the stream. Our experiments with a variety of synthetic and real data sets demonstrate that SSIJ consistently outperforms the state-of-the-art algorithm in terms of the maximum sustainable throughput of the join while being also able to accommodate deadlines on stream tuple processing.**

I. INTRODUCTION

Advances in information technology and business automation have multiplied our potential to generate and analyze huge amounts of data. Business intelligence (BI) has emerged as a set of technologies that enable an enterprise to make better decisions. Data warehousing technology and tools provide an integral part of any decision support system. From an operational viewpoint, the data warehouse is a single, integrated informational store in which the enterprise can evaluate its data over time. Typical data warehouses are updated in a batch, periodic fashion (i.e., every night). Since data may be coming from multiple operational and/or legacy systems across the organization, or even, sometimes, from external sources, significant cleansing, transformation and reconciliation is often necessary before the data is loaded in the repository.

Mihaela A. Bornea was supported by the European Commission and the Greek State through the PENED 2003 programme. Vasilis Vassalos was supported by the European Commission through a Marie Curie Outgoing International Fellowship and by the PENED 2003 programme of research support. Antonios Deligiannakis was partially supported by the European Commission under ICT-FP7-LIFT-255951.

S1

id	desc
1	book
2	DVD

S2

id	desc
1	DVD
2	CD

Surrogate

id	src	gid
1	S1	10
2	S1	20
1	S2	20
2	S2	30

DWH

id	desc
10	book
20	DVD
30	CD

Fig. 1. Surrogate Key Replacement

Extraction-Transformation-Loading (ETL) processes handle this task during the refresh, off-line periods [1]. Many of these ETL processes involve expensive joins between the newly arrived records and some warehouse data or metadata tables. For example, record keys are often replaced with surrogates keys for compactness and consistency. This process, also known as *conforming* [1], necessitates the join of the refresh tuples from each source with a metadata table that relates keys and surrogate keys, as exemplified in Figure 1. Duplicate elimination or identification of newly inserted tuples provide more examples where similar join expressions are encountered [1].

The traditional cycle of updating the data warehouse in a periodic fashion has been exemplified in research and best-practices studies, as it allows us to utilize efficient bulk-loading techniques [2], [3] and avoid interference of the ETL processes with the query workload. However, in emerging applications, such as network monitoring, supply-chain monitoring via RFID technologies and sensory data analysis, the latency introduced from the time that the data is conceived to the time it is ready for analysis may be unacceptably large. Even for traditional business intelligent tasks, finding the right piece of information at the right (i.e., shortest) time is a necessity for survival in today's competitive marketplace. Active data warehousing has emerged as a new BI paradigm where updates from the operational stores are propagated in (near) real-time to the repository. This shift of practices significantly affects the ETL process as the type of joins we described are now between an infinite stream of incoming records and some stored data warehouse table. The output of this operation is a stream which typically participates in additional online operations as part of the repository update. Note that in this context there are no particular tuple ordering requirements.

While there is a lot of research on how to join relations that are either both stored or streaming, there is, surprisingly, little work [4] devoted to the problem of joining a streaming relation with a stored one, as required by an ETL process of

an active data warehouse. The techniques proposed for this problem are extensions of nested-loop joins where (part of) the stream represents the outer relation while the disk-resident relation becomes the inner. The element that differentiates these approaches is the use of an index on the inner relation.

The work in [4] first drew attention to the need to support streaming updates in an active data warehouse. The proposed MeshJoin algorithm utilizes sequential reads from the stored relation in a round robin fashion in order to amortize the I/O look up cost for a large number of stream tuples. In our work we demonstrate, formally as well as experimentally, that pure sequential reads while performing the join are suboptimal both in utilizing efficiently the available memory and in reducing the cost of I/O, when the goal is to increase the supported rate of the stream tuples and, thus, the throughput of the ETL task. MeshJoin also ignores existing indexes in the stored relation that can help speed up the join.

In this paper we contribute by introducing Semi-Streaming Index Join (SSIJ), an index-based algorithm for joining a relational stream with a disk resident relation. SSIJ possesses several characteristics that are critical for its application environment: (i) SSIJ exploits available memory resources to cache frequently accessed pages of the relation, thus being able to quickly join several stream tuples without incurring I/Os for them. The set of cached disk pages is determined based on simple and intuitive statistics; (ii) Accesses to pages (cached or not) of the relation are always batched, in order to minimize the access costs and to take advantage of common relation access patterns of different stream tuples; (iii) Blocks of the relation that are not located in memory are read only if needed (unlike prior techniques like MeshJoin), and based on an optimal plan; (iv) SSIJ dynamically adapts memory allocation between caching relation blocks and storing incoming stream tuples in order to cope with different characteristics of the streamed relation, such as tuple arrival rate, while fully exploiting the available memory for faster processing; (v) SSIJ supports equality as well as range join conditions, works well for arbitrary join relations (one-to-one, one-to-many, many-to-one) and produces the exact join result; (vi) SSIJ is non-blocking, and produces a high rate output stream, allowing this way the kind of pipelined plans essential in data stream management[1]; and last but not least, (vii) SSIJ adapts its execution in order to meet processing deadlines for the incoming stream tuples.

SSIJ is an end-to-end index-based algorithm for stream to relation joins enhanced with efficient batch processing, an optimum disk block retrieval plan and an effective memory management. All additional components are important and as a result of their interaction SSIJ obtains best-in-class performance. Our experiments compare SSIJ to existing specialized techniques and state of the art processing algorithms used in relational DBMSes. We also explore the parameter space in order to shed light on SSIJ performance and to tune its parameters. Our analysis demonstrates that SSIJ consistently outperforms MeshJoin in terms of the maximum sustainable throughput of the join, for a variety of synthetic and real data sets. A formal analysis of why this occurs, even in the worst case of our algorithm, is presented in Section V. We also show that state of the art implementations of index-based relational techniques are not suitable for this problem.

The rest of the paper is organized as follows. Section II presents the related work. In Section III we discuss the characteristics of the index our algorithm uses. Section IV introduces SSIJ, while Section V presents a comparison with MeshJoin. Section VI presents our experimental study, while Section VII contains concluding remarks.

[1]Of course, for traditional pull-based pipelined plans the output of any stream operator, including SSIJ has to be buffered.

II. RELATED WORK

Increasing the efficiency of the join operator has been an important topic of research for the databases community. Surprisingly, little attention has been paid to setups in which one relation is disk-resident while the second is streamed.

The MeshJoin algorithm [5], [4] presented in Section I is the most relevant piece of existing research work. The authors demonstrated that MeshJoin outperforms traditional join algorithms and achieves a higher throughput of the join operator. In Section V we provide an extended comparison of MeshJoin with SSIJ.

The work in [6] focuses on reducing the update propagation delays introduced by ETL processing rather than increasing the maximum supported rate of the join. The main idea is to create hash or range-based partitions for the relation and to organize accordingly an in-memory stream wait buffer. However, infrequent stream tuples can wait for long periods, even longer than with MeshJoin. In contrast, as we demonstrate in the experiment in Figure 16, SSIJ introduces significantly smaller delays and is able to adjust its execution in order to meet stream processing deadlines.

Active data warehouses [7], [8], [9], [10] have appeared as a new data management paradigm where updates from operational stores are propagated in (near) real time to the repository. Active data warehouses require a fundamental shift in the design of Extraction-Transformation-Loading (ETL) processes, since most prior work in the area, including, for example, the detection of duplicates and the surrogate key replacement [11], [12], [13] have assumed a batch, off-line refresh of the warehouse.

Traditional query processing has explored variations of some of the techniques incorporated into SSIJ. [14],[15],[16] present improvements of index nested loops included in commercial database systems. These papers consider techniques like caching, prefetching or the partial sort of the outer relation in order to increase the locality of reference. The problem of finding an optimum read schedule for a set of disk pages has also been addressed in [17]. We detail our contributions with respect to [17] in Section IV-D. While at an abstract level all the individual aforementioned techniques propose solutions to increase the efficiency of the join operator in specific ways, our work is the first to provide a complete algorithm and a

thorough analysis of how to offer an efficient solution to the problem of stream to relation join. Specifically, as we show in our experimental study, SSIJ vastly outperforms join operators provided by traditional relational systems.

The importance of a stream to relation join operator has also been acknowledged in general purpose DSMS systems like Telegraph [18], Gigascope [19], Aurora [20], STREAM [21]. One of the open issues in these systems refers to servicing queries involving streams and historical data from disk. An approximate solution proposed to this problem was included inside the Telegraph project. OSCAR access method [22] proposes addressing the problem of joining streamed sensor data with "history" tables by retrieving only a reduced version of the data on disk when the system is overloaded. Reduced versions of the history are retrieved for correspondingly fewer I/Os. Unlike OSCAR, we focus on providing exact join results. OSCAR or load shedding techniques [5] can be used in addition to SSIJ to deal with exceptional cases of extremely high stream arrival rates.

III. Preliminaries-Index use

Our algorithm assumes and makes effective use of an index on the join attribute of the relation. The only requirements from the used index is that it can return a set of block ids (or offsets of these blocks) where matching (i.e., joining) records of that key can be found. To implement such an index, many widely used techniques can be considered, such as B^+trees [23], hash tables and bitmapped indexes [24] (for the latter the record pointers are implied by the location of the 1s in the bitmap). The B^+tree has the advantage of efficiently supporting range queries.

In our implementation we use a B^+tree for indexing the relation. Furthermore, relation tuples themselves are stored at the leaves of the B^+tree. (In our discussion these leaves are referred to as *relation blocks or relation pages*.) Such a structure has the benefit of reducing the indexing overhead, typically by reducing the height of the tree by one level, thus allowing the non-leaf nodes of the index to more easily fit within the available memory. Moreover, it is widely implemented in commercial database systems, under different names (e.g., "Index-Organized Table" in Oracle, "Clustered Index Table" in SQL Server, etc). However, using such an index/relation organization is not necessary for SSIJ which can just as easily use any index with a memory footprint similar to the B^+tree.

For ease of presentation, we assume that the non-leaf pages of the index are brought into memory at the beginning of the join and remain pinned until the join completes. Index nodes do take up space in main memory, which is accounted for in our memory budgeting. In our index of choice, fitting the upper levels of the B^+tree (and not the relation, stored at the leaves) in memory is expected to be possible for most realistic application scenarios mentioned in Section I. More generally speaking, in the experiments presented in this paper, the size of the internal nodes of the B^+tree index is no more than 13MB for a relation size equal to 10GB.

Symbol	Description
DR	Disk-resident relation
SR	Streaming relation (stream)
CR	Memory cached blocks/pages of DR
IB	Input buffer for unprocessed stream tuples
SB	Stream buffer: stream tuples waiting for disk blocks to be read for their join
IB_{thresh}	Minimum number of tuples in IB for the online phase to kick in
SB_{thresh}	Minimum number of tuples in SB for the join phase to kick in
p	A page of DR (cached or not)
s	A tuple of SR
II_p	Inverted index list for relation page p. Contains pointer to matching tuples in SB
M	Total memory allocated to SSIJ
P_u	Set of blocks that SSIJ must read in its *join phase*
S	Disk average seek time
T	Disk average transfer time of a disk page
$maxDist$	Maximum distance (in disk pages) of two pages b_1 and b_2 such that reading with one sequential I/O all blocks from b_1 to b_2 is cheaper than reading only b_1 and b_2 with two random I/Os
$maxScan$	Maximum length sequence of disk blocks that SSIJ may read with a single sequential scan

TABLE I
Main Symbols Used

Pinning these index pages is not essential for the operation of our algorithm. Under more constrained memory environments, these index pages can be replaced using the same cache replacement policy as with the pages containing the tuples of the relation. In such a case, retrieving needed index pages on demand during the algorithm operation imposes an overhead, but this overhead is fully amortized due to the batch processing of input stream tuples (described in Section IV-B).

Our discussion of the algorithms is based on our selected index. While several parts of the algorithm remain unaffected regardless of the used index (i.e., the in-memory online phase, the plan generation for the join phase, the dynamic memory allocation or the cache replacement policy), presented optimizations (i.e., the batched traversal of the index) are specific to our index of choice. Moreover, during the online phase, the algorithm performs a sort operation based on the characteristics of the used index. For the case of our B^+tree index, this corresponds to sorting based on the join attribute. For other types of indexes, this sorting operation depends on the grouping characteristics of the index. This sorting process allows sharing index scans between several stream tuples.

IV. SSIJ Framework

We now present the SSIJ algorithm. Table I summarizes some important notation used throughout this paper.

A. Dynamic Memory Partitioning and Used Data Structures

We keep a detailed account of memory usage. The available memory M for the join operator is partitioned in our algorithm into five disjoint sets, whose size is not fixed, but rather dynamically adjusted during the operation of our algorithm. These five sets include the index, the set of cached relation blocks, two buffers regarding stream tuples, and an inverted index. We now describe these sets in more detail while the relation between these components is pictured in Figure 2.

The index. Our algorithm makes effective use of an index on the join attribute of the relation.

Cached relation blocks CR. This part of memory contains

blocks of the relation (i.e., leaf pages in our $B^{+}tree$ index) that have been brought from disk to be joined with the stream. A utility counter is kept for each block and is used by the caching policy, as explained later in this section.

Input buffer IB**.** Stream tuples that have arrived, but have not started their join operation, are placed in IB.

Stream buffer SB**.** Any tuple that we check using our index, and determine that this tuple requires retrieving one or more blocks of the relation from disk, is stored in the stream buffer SB. These required relation pages will be read, using a plan that the algorithm determines, when SB_{thresh} tuples have accumulated in SB. The size of the stream buffer is a parameter of SSIJ, and its impact is experimentally evaluated in Section VI.

Inverted Index II**.** For each relation disk block that needs to be read from disk (because some stream tuple in SB required its presence in memory), we maintain a list with the location of all matching stream tuples in SB for it. Multiple uses of the inverted index exist. Besides improving the performance of the join phase, the index is also important for efficiently guaranteeing the correctness of the overall process of SSIJ. A third use of the inverted index will be shortly evident, when we describe the cache replacement policy.

B. Stream Processing Algorithm

Overview. The algorithm consists of three phases, namely the *pending*, *online* and *join* phases. In a nutshell, in the pending phase the algorithm waits for a minimum of IB_{thresh} tuples to accumulate (or until it receives an END_OF_STREAM message) before it moves to the online phase. This occurs in order to batch process the incoming stream tuples. Compared to a naive approach that processes individual tuples one-by-one, a batch process allows SSIJ to take advantages of common access patterns, which in turn amortizes index and cache lookup cost. This claim is also supported by the experimental results in Figure 12. In the online phase, stream tuples from the input buffer IB are looked up using the index (Sequence 1 in Figure 2) and joined with in-memory (cached) blocks containing relation tuples (Sequence 2). Stream tuples whose join is not completed in the online phase (i.e., they require relation blocks that are not cached) need to wait for the join phase, when the corresponding disk blocks are read using a read plan that our algorithm generates. In Figure 2 the matching page p_i of tuple t_i is found in CR and the join of this tuple is completed during the online phase (Sequence 3a). Since the matching relation block of tuple t_k is not present in CR, t_k is placed in the stream buffer (Sequence 3b). The join of t_k is completed in the join phase when its matching relation page p_k is loaded from disk (Sequences 4,5,6). When the join phase is completed, the algorithm again moves into the pending phase.

The main ideas of the SSIJ algorithm are: (1) Batch stream tuple processing: This involves not only batching the disk reads required by stream tuples, but also batch index lookups; (2) Fast joining read (relation) disk blocks with matching

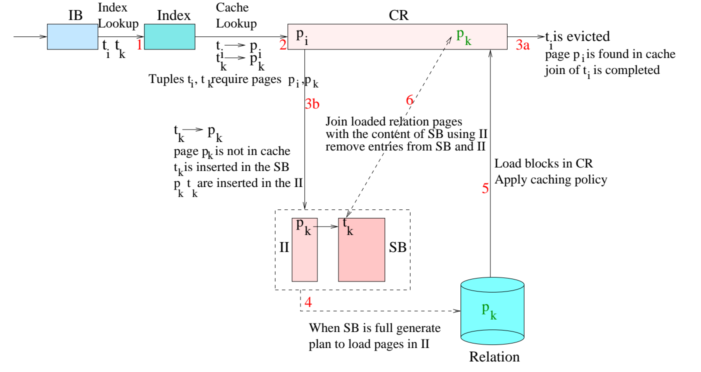

Fig. 2. SSIJ Overview

Algorithm 1 OnlinePhase

1: Sort the first IB_{thresh} based on the characteristics of the used index
2: Batch scan the index using the sorted sequence, in order to locate the set MB of matching (joining) relation blocks
3: **for** each page $p \in MB$ already cached **do**
4: Perform join of p with IB_{thresh} and output result tuples
5: Increment the utility counter of p by the number of stream tuples it joined with
6: **end for**
7: **for** each stream tuple s **do**
8: Let MB_s denote the list of the matching block ids for s, located from the index
9: **if** ALL pages in MB_s are in CR **then**
10: Discard s
11: Go to line 7
12: **end if**
13: **for** any page $p \in MB_s$ and NOT in CR (not cached) **do**
14: Insert into II_p a pointer to s
15: **end for**
16: Insert s in stream buffer
17: **if** not enough space and cached pages exist **then**
18: Remove from cache the page with smallest utility counter
19: **end if**
20: **end for**
21: **if** SB_{thresh} tuples exist in SB **then**
22: Initiate join phase
23: **else**
24: Go to pending phase
25: **end if**

stream tuples; (3) Maintaining in memory the disk pages that are requested more often, and quickly determining which pages to maintain; (4) Reading only areas of the relation that are requested, using an optimal plan that uses a mixture of sequential and random I/Os; and (5) Dynamically adjusting the memory of the data structures presented in Section IV-A.

For the input buffer, the stream buffer and the inverted index, we initially reserve one page for each. In the remaining memory we first load the index (or whichever part of it fits, in constrained cases) and then we reserve the rest of the blocks for the relation.

Online Phase. (Algorithm 1) When the online phase kicks in, the first IB_{thresh} tuples that have accumulated into IB are sorted based on the characteristics of the used index, as explained in the previous section (Line 1). This sorting allows us to share scans of the index and of the cached relation pages among several tuples (Line 2). For all matching relation blocks/tuples that are in the cache, the join result is output immediately (Lines 3-4), so that, if the join operator

Algorithm 2 JoinPhase

```
 1: Apply caching policy. Determine ToKeep and ToRemove lists.
 2: Sort matching block ids P_u (e.g., based on disk offset)
 3: Generate read plan RP using DP algorithm
 4: while RP contains more sequences to read do
 5:    if not sufficient space for next block sequence then
 6:       Evict appropriate number of cached pages using ToRemove
 7:    end if
 8:    Read next block sequence SN
 9:    for all blocks b in SN do
10:       Join b with stream buffer SB using inverted index
11:       if b not in ToKeep then
12:          Evict b from cache
13:       end if
14:       Drop from the II the entries related to b
15:    end for
16: end while
17: Flush SB
18: Clear utility counters in cache
```

is part of a pipeline, we do not stall the pipeline. The utility counter of any page in CR is increased by one for every stream tuple of IB_{thresh} that it joins with (Line 5). For each stream tuple s, let MB_s denote its matching blocks (which were located at Line 2). If *all* the matching relation blocks for s are in the cache, then the join for s is complete, and s can be discarded (Lines 9-10). If some matching relation blocks for s are not in the cache, the join with these blocks is not performed immediately, as this would mean that the algorithm would pause until these blocks are fetched from disk. Instead, the SSIJ algorithm will process the join of s with the disk resident matching blocks of the relation at a later point, during the *join phase*, in order to better amortize the cost of required I/O among several stream tuples. Since, at this step, we have identified the matching disk blocks for s, we record this information by updating the inverted index, in order to speed up the join computation when these disk blocks are later retrieved from disk (Lines 13-15). These disk blocks form the set P_u of blocks SSIJ must read from disk in the join phase. Moreover, s is stored in the stream buffer SB (Line 16). If there is insufficient space for this operation, but cached pages do exist, then the cached page with the smallest utility counter is evicted (Lines 17-19). CR improves the performance of the join and unlike other components of the algorithm (i.e. IB, SB) its content is not necessary for result correctness.

After the batch of IB_{thresh} has been processed, the algorithm may move to the join phase, if SB_{thresh} tuples exist in the stream buffer (Lines 21-22), or switch back to the pending phase (Line 24).

Join Phase. (Algorithm 2) We now present a high level overview of the algorithm for the join phase of SSIJ. Due to the multiple operations contained in the join phase, our initial presentation will proceed assuming the correct functionality of two of these operations, namely: (a) The operation that generates the read plan of disk pages to be fetched from disk (Line 3); and (b) How the cache eviction algorithm works (Lines 1,6). These two operations will be initially presented as *black boxes*, and their details will be introduced immediately after the overview of the join phase.

During the join phase, we join all stream tuples in the stream buffer SB with their non-cached matching relation pages P_u. A key part of this process is how to formulate a plan for reading the requested matching disk blocks. More specifically, we need to determine whether the required disk blocks will be read individually using random I/Os, or in larger sequences using sequential I/Os. While we defer the details of this plan generation process for Section IV-D, this plan generation process requires that the input P_u pages be sorted based on their physical layout; in the simplest case, this corresponds to sorting the offsets of the relation blocks on disk. This is performed in Lines 2-3 of the algorithm.

It is important to note that the read plan may cause some disk pages to be loaded in spite of the fact that they are not requested by the stream, as part of a sequentially loaded disk segment that amortizes the I/O cost. Such "unwanted" disk blocks that are read because of sequential I/Os are evicted from cache immediately, since they have zero utility for the join (i.e., zero utility counters).

During the join phase, the algorithm continuously reads (Lines 4-16) sequences of disk blocks (based on the generated read plan). Each sequence of read relation blocks, as directed by the generated read plan, is fetched from disk and is inserted into the cache, replacing those cache pages with the lowest utility counters (Lines 5-7). The details of the replacement policy are presented in Section IV-C. The read disk blocks are then used to generate output join tuples by joining with the appropriate stream tuples, using the II (Line 10). After the join of the sequences is complete, the corresponding entries in the II can be safely removed (Line 14). On the other hand, each joined relation page is evicted or not (Lines 11-13) based on the decisions of the cache replacement policy (Line 1). Finally, after all the disk blocks in P_u have been read and joined, the processed stream buffer tuples are flushed from memory (as their join is complete), and the utility counters for all pages in the cache are reset (Lines 17-18).

C. *Cache Replacement Policy*

There are three situations when a page from the cache needs to be removed/replaced. The first case is when new stream tuples arrive during the online phase and need to be accommodated in the input buffer. In this case, if there is insufficient space, the cached page from the relation with the lowest utility counter is evicted.

The other two cases occur during the join phase. On one hand, when a new sequence of blocks is read from the relation, if there isn't enough space in the cache, the appropriate number of cached blocks with the lowest utility is evicted. On the other hand, stream tuples continue arriving during the join phase, and if there is insufficient space to store an incoming tuple, the algorithm dynamically evicts the relation page with the smallest utility *as long as it's not in the block sequence currently being joined.*

An efficient implementation of the cache replacement policy during the join phase is far from trivial. A naive approach that maintains a sorted list of the page ids in the cache based on their utility counters, and on demand flushes the

pages with the lowest utility counters, incurs a high cost. In particular, the repeated insertions in the list can introduce a large overhead when the amount of memory devoted to SSIJ is large. Moreover, during the online phase, the utility counters of the cached blocks are intensively updated, which requires continuous reorganization of the sorted list. Maintaining the cache blocks in a priority queue exhibits similar problems.

Our implementation is based on the key observation that we do not need to actually read a page in P_u in order to calculate its utility counter, as we have all the necessary information in the inverted index of each page. Each entry in the II associates the id of a page that needs to be read from disk with the list of pointers to matching stream tuples in SB. Thus, the utility of a page in II is equal to the number of elements in the list it is associated with.

So, at the start of the join phase we know the utility counter of (i) all pages currently in the cache, and (ii) all pages P_u that will be read in the join phase. At this point we have enough information in order to start making eviction decisions (Line 1 of Algorithm 2). More specifically, we sort the ids of the blocks in CR and the ids of matching disk pages P_u based on their utility. Considering the available memory (i.e., memory after subtracting the space needed for the input buffer, the stream tuples and the index structures), we determine the sorted subset, denoted $ToKeep$, of pages from P_u that should be in cache at the end of the join phase, given their utility. We also determine the sorted subset, denoted $ToRemove$, of block ids that are already in the CR and are going to be replaced by blocks in $ToKeep$ with higher utility. If, during the join phase, the size of the cache is reduced due to the arrival of stream tuples, we can correspondingly increase the size of $ToRemove$ or reduce the size of $ToKeep$ according to the utility of pages in these lists. Whenever in Lines 5-7 of Algorithm 2 we evict some pages, we can simply evict *any* of the pages in $ToRemove$: these pages are not needed for the rest of the join phase and, given their utility, will not be in cache at the end of the join phase.

D. Generating An Optimal Read Plan

Basics and Notation. Let P_u denote the set of matching disk pages that our algorithm wishes to read from disk during the join phase. Let S denote the average seek time to locate the start of a random disk block, and let T denote the time to transfer one block into memory. Then the cost of reading N disk blocks using sequential I/O is: $C(N) = S + T \times N$.

Assume that we want to read two disk blocks B_1 and B_2 that physically lie $dist$ blocks apart, but we are not interested in reading any one of the disk blocks in between them. Based on our cost model, the cost of reading B_1 and B_2 using two random I/Os is $2(S+T)$, while the cost of reading them using a single sequential scan is $S+T\times(dist+1)$. Thus, a sequential scan is preferable if $dist < 1 + \frac{S}{T}$. In our discussion hereafter we use $maxDist = 1 + \frac{S}{T}$ to denote the maximum distance in disk blocks between two consecutive pages in P_u where it would make sense (cost-wise) to combine these pages in a common sequential scan.

All data read by a single sequential scan need to be brought into memory, thus forcing a portion of the cache to be evicted. In order to avoid flushing a large part of the cache, it seems natural to impose a maximum size $maxScan$ for the sequential scan.

Prior work on this problem. Seeger et al in [17] formulated this problem as a single-source nearest neighbor problem, which can be solved by any nearest neighbor algorithm. Due to the large running time of such algorithms (i.e., Dijkstra's shortest path in a graph with $|P_u|$ nodes and $|E|$ edges requires $O(|P_u|log|P_u| + |E|)$ time, while in this problem $|E| = O(|P_u| \times maxScan)$), [17] also proposed a greedy algorithm for the same problem. In a nutshell, the greedy algorithm tried to read the desired set of blocks in maximum sequences of read blocks. Each such sequence was picked as: (i) to not violate the maximum number of read blocks at each time, and (ii) to not read in any sequence more than k-continuous unwanted disk blocks (i.e., blocks not in P_u), where k is a parameter of the algorithm.

An important observation that [17] made is that the optimal plan for retrieving the matching relation blocks consists of block sequences whose first and last blocks are guaranteed to be pages in P_u. Moreover, one can also demonstrate that the optimal plan cannot allow overlaps in its read sequences of blocks (i.e., no block will be read twice by the optimal plan).

Our Contributions. The greedy algorithm of [17] performed well in practice, but does not have a guaranteed approximation factor. On the other hand, the running time complexity of the optimal algorithm of [17] is prohibitive. We thus seek to devise a more efficient optimal algorithm for the same problem. Our $O(|P_u| \times MaxScan)$ optimal DP algorithm requires a single array of $O(|P_u|)$ space to operate, thus yielding an efficient implementation. This improves upon the corresponding complexities of the optimal algorithm in [17]. Of course, since this is just one of the components of our algorithm, one can also choose to use the algorithms of [17].

Our DP-Algorithm. Let P_j denote the j-th page in the ordered (based on their offsets) list of disk pages P_u. Our optimal DP algorithm computes the minimum cost $OPT[P_j]$ for reading all pages, in multiple sequences, in the list P_u starting from P_j. We will consider all possible ways of reading P_j (starting from it), which equates to all possible sequential scans of length $1, \ldots, MaxScan$ that have as their last page a page in P_u, and then optimally reading the remaining pages in P_u that were not covered by this scan. Let $P_{end(P_j,i)}$ denote the last page of a scan of i continuous disk blocks starting from P_j. Then,

$$OPT[P_j] = \min_{\substack{1 \leq i \leq MaxScan \\ P_{end(P_j,i)} \in P_u}} \{S + T \times i + OPT[nP_{end(P_j,i)}]\}$$

where the notation $nP_{end(P_j,i)}$ refers to the first page in P_u after $P_{end(P_j,i)}$. Since entries of the dynamic programming solution are calculated only for sequences ending at disk blocks in P_u, the overall running time requirements are $O(|P_u| \times MaxScan)$ and the space requirements are $O(|P_u|)$.

E. Handling Updates

In many of the settings for the use of SSIJ the stored relation contains historical information, and hence it is not expected to be updated. Nevertheless, in other setups, updates, insertions and deletions may happen to the relation. In order to accommodate updates, SSIJ relation blocks have a relatively low *filling factor*. The impact of the filling factor on SSIJ performance is discussed in Section VI-D. Updating $B^{+}trees$ and other index structures has been studied in detail, e.g., [25].

Update Semantics. A correct result in the presence of updates in the relation is obtained when each stream tuple sees a consistent view of the database: either the relation instance before or the relation instance after an update is used. The effect of these semantics is equivalent to considering that the join of a stream tuple with the relation forms a transaction. The semantics we propose ensure robust transactional properties between stream tuples and the relation. Maintaining transactional properties, like isolation, when updating a database table while it is concurrently being joined with a continuous stream of tuples has been identified as an open research problem in the stream processing community [26]. Currently there are no other generally accepted semantics that would ensure transactional properties for the entire stream.

Installing Updates. Before SSIJ allows processing any incoming updates, the join phase is forced. This prevents result inconsistencies in situations where a stream tuple has already produced results during the online phase (using the relation instance before the pending update) but it requires, in order to complete the result, a disk block which might be modified by the pending update before the next join phase is executed. While installing updates, stream tuples are inserted in the IB and SSIJ processing is suspended until the update is installed completely. At this point SSIJ can resume processing or can allow installation of the next pending update, depending on the number of stream tuples accumulated in the IB. It is obviously beneficial to allow updates execution during periods of low stream arrival rates.

F. Handling Deadlines

Many streaming applications may require that the effect of a stream tuple is visible in the output before a given time limit. This time limit is associated with each stream tuple in the form of a deadline. We now present the minor modifications to SSIJ in order to consider deadlines.

The stream tuples that are processed during the online phase are selected from the IB in increasing order of their deadline value. We maintain the closest deadline for the tuples in the IB. If fewer than IB_{thres} stream tuples are available in the IB, the online phase might be triggered in order to prevent deadline expiration. In a similar manner, we account for the maximum latency that can be tolerated by a stream tuple in the SB and we use this value to prompt the join phase.

In the presence of deadlines the online and join phases are triggered using a simple cost model that associates the number of blocks to be loaded from disk with their processing time. Thus, when the tuples in the SB target a number of relation blocks for which the cost model indicates a processing time close to the maximum latency, SSIJ triggers the online phase. During the online phase the tuples in the IB whose deadline would expire (based on the estimated join time by the cost model) while SSIJ performs the join phase are processed. [2] Initially the cost model assumes that for each loaded block SSIJ performs a random I/O operation. This initial model is further refined during the algorithm's operation based on corresponding worst-case costs of previous join phases for a similar number of retrieved blocks.

Stream latency awareness is a characteristic of SSIJ which is not shared by the main other existing techniques. We could not think of simple ways of making MeshJoin deadline aware.

V. Comparison to MeshJoin

Worst Case Throughput. In order to estimate the throughput of our algorithm, we need to consider the possible ways that an incoming stream tuple is processed, while checking for matching tuples from the relation:

- For those matching tuples from the relation that exist in the cache, the stream tuple is joined immediately with zero I/Os.
- If there exist matching tuples from the relation that are not in the cache, they need to be retrieved from the disk.

As described in Section IV, in our algorithm disk blocks are not retrieved for every incoming stream tuple, but for batches of stream tuples, in order to better utilize:

- Common access patterns: The matching tuples of several stream tuples may be located on the same disk block. The disk block will be read just once for each batch.
- Physical location of read disk blocks: Using the simple but precise cost model presented in the previous section, SSIJ decides whether it is more beneficial to read disk blocks that are not stored "very far apart" using sequential scans, or with multiple random I/Os.

The worst case throughput of our algorithm will occur if (i) No stream tuple is processed using the cache; (ii) Each stream tuple needs to retrieve different disk blocks from the other stream tuples (i.e., no overlap exists in the blocks that need to be read); and (iii) The retrieved disk blocks are "as far as possible" from each other (we will analyze this further in the next paragraph). The latter condition implies that the maximum distance is at least $1 + \frac{S}{T}$. Such a distance or greater will either force our algorithms to read all disk blocks sufficiently far from each other using random I/Os or, otherwise, to read multiple "not useful" disk blocks with our sequential scans.

Let $B(DR)$ denote the number of disk blocks spanning the relation DR. Let the number of stream tuples batched together in the stream buffer be $Tup(SB)$ (this number is at least equal to SB_{thresh}, but may be higher, since we move to the join phase after an online phase, and not immediately when the

[2] Note that during the online phase the algorithm does not perform I/O operations and the delays that are introduced are negligible.

SB_{thresh} tuples are accumulated). The average distance of consecutive useful disk blocks containing matching tuples for the stream tuples is then $\frac{B(DR)}{Tup(SR)}$. Recall that we denote by $maxScan$ the maximum number of blocks that can be read by a single sequential scan. Thus, the worst case distance of consecutive useful disk blocks containing matching tuples is $\frac{B(DR)}{maxScan}$. Comparing this ratio to $maxDist$, the worst case throughput $WCThr$ of our algorithm is:

$$
\begin{aligned}
WCThr &= \frac{Tup(SB)}{\min\{Tup(SB) \times (S+T), \frac{B(DR)\times S}{maxScan} + B(DR) \times T\}} \\
&= \max\{\frac{1}{S+T}, \frac{Tup(SB)}{\frac{S\times B(DR)}{maxScan} + B(DR) \times T}\} \\
&= \max\{\frac{1}{S+T}, MeshJoinThroughput\}
\end{aligned}
$$

The latter equality is produced by setting $maxScan$ equal to the number of chunks read sequentially by MeshJoin. It is important to note that in the worst case our algorithm will exhibit the throughput achieved by MeshJoin. Of course, the average case may be much better. For example, the stream tuples may exhibit temporal locality, thus requiring only a small part of the relation to be read (in contrast to the entire relation for MeshJoin). Moreover, the cache can help significantly increase the throughput by reducing the number of stream tuples for which we need to retrieve their matching blocks from disk. Finally, due to the dynamic allocation of memory between the cache pages and the stream tuples, our algorithm can better adapt to cases of sudden bursts (i.e., increased rate) from the stream. On the contrary, MeshJoin uses a fixed memory budget for both the steam and the relation. Our experiments validate the above analysis.

Intelligent Cache Management. Both SSIJ and MeshJoin make use of the available memory for storing stream tuples as well as parts of the relation. In MeshJoin, the algorithm reads blocks of the relation (in a sequential round-robin manner) and uses the records in these blocks to "probe" (via a hash) the part of memory that contains the stream tuples. Thus, MeshJoin essentially "caches" the stream, while using the relation to query this cache for records that join, in a data oblivious manner. In our algorithm, the situation is reversed. The stream tuples are used to probe (via the index) the relation in search of records that join. Thus, we choose to cache the relation blocks that are retrieved, in hope that these blocks may get reused in future probes by the stream. Given this simplified, high-level view of both algorithms, one can observe that MeshJoin is forced to pin stream tuples in memory, while waiting for the table scan to complete, otherwise, it will not be able to compute the exact result of the join. In SSIJ, on the other hand, we only need to have in the cache the relation blocks that were requested by the currently processed stream buffer tuples[3]. This allows us to choose a smart caching policy that can be expected to maximize the join throughput. In a nutshell, for MeshJoin, caching is dictated by the algorithm operation and caching decisions cannot be tuned or improved, while in SSIJ intelligent caching can be used to improve performance.

[3]Per Section III, keeping upper level index nodes also helps.

Parameter	Default Value
Index used	B^+tree with tuples stored at leaf nodes
Join key domain size	Integer in [0,250000]
Key value distribution in DR	Uniform
Key value distribution in SR	Zipfian
Size of relation	10GB
Number of stream tuples	20M
Filling factor of data pages	80%
Disk block size	8192 bytes
Stream buffer size	400000 for memory = 1% of the relation 1000000 otherwise
Input buffer size	250000
Tuples per block	64
MaxScan	200

TABLE II
DEFAULT PARAMETERS IN EXPERIMENTS

VI. EXPERIMENTS

In this section, we present an extensive experimental study of SSIJ in a direct comparison to the existing MeshJoin algorithm, over several real-life and synthetic data sets when varying a large number of parameters. We also present an analysis of the impact of various algorithm parameters on the performance of SSIJ. Additionally, we provide a comparison of SSIJ with index join algorithms supplied by commercial and open source database systems.

Methodology, Techniques and Parameter Settings. The throughput rate is a raw performance metric that indicates how many stream tuples can be processed in the time unit. In order to measure the throughput rate for both SSIJ and MeshJoin, we use a constant arrival rate for the stream tuples throughout a run and try executing the join using each algorithm for the generated stream. The reported numbers indicate the maximum rate that each algorithm can handle, by repeating the experiment with gradually increasing arrival rates, without memory overflow. For MeshJoin we used the code provided to us by the authors of [5]. Table II presents the default parameter settings in our experiments. The parameters of the MeshJoin algorithm were computed as described in [5].

Data Sets. We experimented with synthetic and real data sets and present in this section a representative set of results. A detailed description of the synthetic and real data set is presented below.

Synthetic Data. For the synthetic data, the join attribute values of the data stream follow a zipf distribution. We present results for several skew values, varying from 0 (corresponding to a uniform distribution) to high skew values. We also experimented with different shapes of the zipfian distribution. Each zipfian distribution is assigned one of 2 possible shapes: (1) "NoPerm" is the typical zipfian distribution, where smaller domain values are assigned higher probabilities; and (2) "Random" is a zipfian distribution, where the domain values are randomly permuted resulting in placing the hot-spots of the distribution at random places over the domain.

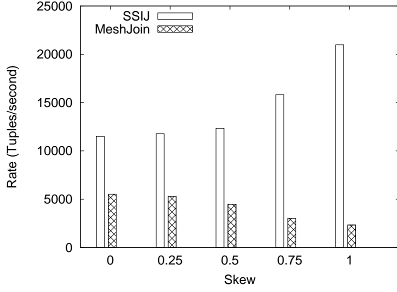

Fig. 3. Throughput, NoPerm Zipf

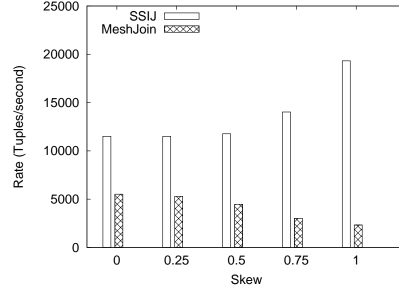

Fig. 4. Throughput, Random Zipf

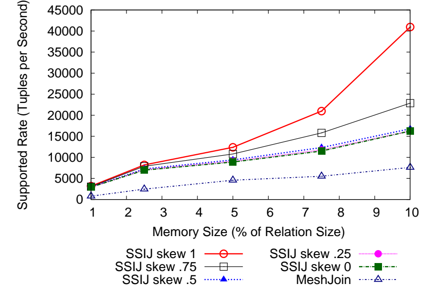

Fig. 5. Throughput, NoPerm Zipf

Temporal Locality. Several features of our SSIJ algorithm have been evaluated for data sets exhibiting temporal locality. In these data sets the join values of the stream tuples within a time period fall mostly within a subset of the relation domain. We created such a data set as following: The relation contains 80 million tuples uniformly distributed from 0 to 5 million. The stream contains 40 million tuples and it is split in 20 intervals of 2 million tuples. Each interval i, $i \in [0, 19]$ has a zipf distribution with skew 1 in $[i*250,000,(i+1)*250,000]$. Thus, each stream interval requires localized reads on disk limited to a number of blocks equal to 5% of the relation.

TPC-H. We also experimented using the TPC-H data set, which we created using a scale factor of 100. The data generator that we used is available at `http://www.tpc.org/tpch/`. TPC-H is a well known decision support benchmark. More precisely, table Part, which describes products, and table LineItem, which contains order details are used. In table LineItem there is an entry corresponding to each product included in every order. The Part table has 20 million records. We used the Part relation as the disk resident relation, while the first 10 million tuples of the LineItem relation were used as the streaming relation. The two relations are joined on the *product id* column. The goal of such a join would be to augment the sales data with the product description, for each product used in the order, before adding it to the warehouse.

Weather Reports. Finally, we also used a real data set containing cloud information stored in summary weather reports (this data set was also used in [5] and is available at: `http://cdiac.ornl.gov/epubs/ndp/ndp026b/ndp026b.htm`). We first created the disk resident relation by combining meteorological data corresponding to months April and August, while the stream data was extracted by combining data files from December. The disk relation contained 20 million tuples, while the streaming relation contained 6 million tuples. The tuples in both the disk based relation and the stream were 128 bytes long. Tuples from the stream and the relation joined if they both corresponded to the same longitude measurement and the value domain for the join attribute is the interval [0,36000].

Main Findings. The main findings of our study can be summarized as follows: (1) SSIJ supports high throughput rates; (2) The throughput increases as the available memory increases; (3) The throughput increases with more skewed streaming data; (4) SSIJ substantially outperforms MeshJoin and state of the art techniques included in relational engines; and (5) The sensitivity analysis confirms our theoretical analysis in Sections IV and V and indicates suitable values for SSIJ parameters by exploring their domain.

All experiments reported in this section were performed on a personal computer with an Intel Core i7 processor clocked at 2.80GHz and with 8GB of memory running Linux (kernel version 2.6.31-15). The code for both algorithms was compiled using the gcc 4.4.1 compiler. All the I/O operations on the disk resident relation, for both algorithms, are performed using raw I/O in order to bypass the operating system buffering.

A. Synthetic Data Sets

Sensitivity to Data Skew. In this set of experiments we vary the skew of the zipfian distribution of the join attribute values that appear in the streaming relation from 0 (uniform distribution) up to a high skew value of 1. The available memory was set to 7.5% of the stored relation size. In Figures 3 and 4 we plot the maximum obtainable throughput for the NoPerm and Random shapes of the zipfian distribution and for different levels of skew. The SSIJ algorithm increases its throughput significantly and it behaves equally well with the randomly permuted distribution, which is a very challenging data set. The throughput of our algorithm increases significantly, due to the increased cache hit ratio. On the contrary, the throughput of the MeshJoin algorithm does not change much with the increasing skew, since MeshJoin does not exploit in its algorithm the data skew (its performance actually degrades slightly, which is consistent with the findings of its authors).

Sensitivity to the Available Memory. In Figures 5 and 6 we repeat the previous experiment but vary the size of the available memory from 1% to 10% of the relation size and for different values of skew. For MeshJoin, the only presented line is for skew=0 (which, as indicated by Figures 3 and 4, represents its best case). We note that the SSIJ algorithm takes advantage of the available memory for caching additional blocks of the stored relation and achieves up to 6 times higher throughput than MeshJoin. SSIJ considerably improves its performance as the value of the skew parameter increases. What is equally important though is that the rate supported by SSIJ even in seriously memory constrained environments (close to or more than 3000 tuples per second for 1% available memory) makes it the only feasible solution for a wide range of existing and future applications.

1:K Joins. In this experiment we control the number K of

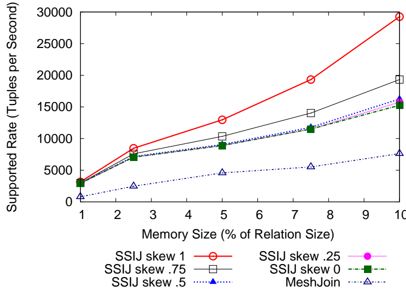

Fig. 6. Throughput, Random Zipf

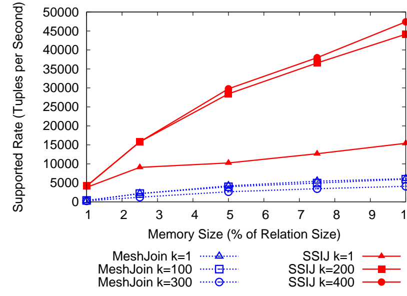

Fig. 7. Testing 1:K Joins

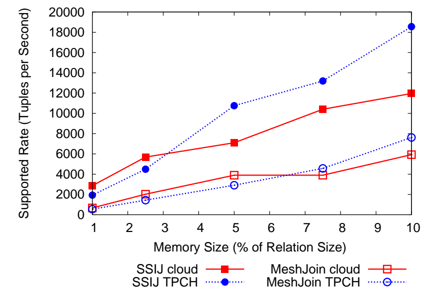

Fig. 8. Throughput, TPC-H and Cloud Data

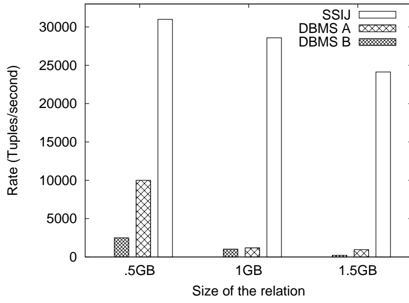

Fig. 9. Comparison with DBMS

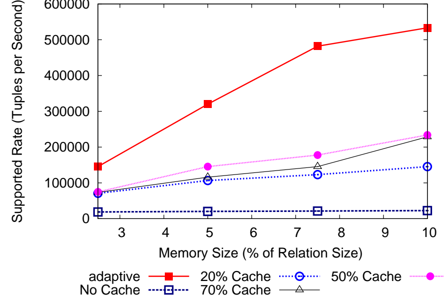

Fig. 10. Relation-Stream Memory Allocation

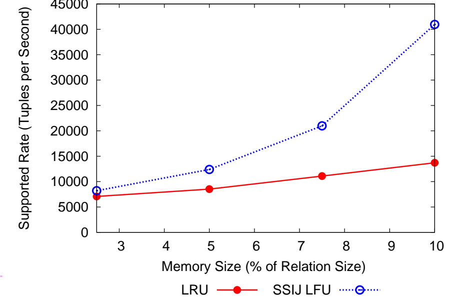

Fig. 11. Evaluating the Caching Policy

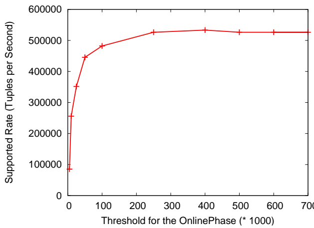

Fig. 12. Sensitivity to IB_{thresh}

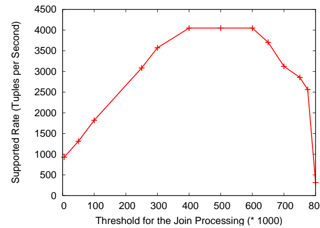

Fig. 13. Sensitivity to SB_{thresh}

the stored relation tuples that join with one stream tuple. The disk relation contains 80 million tuples. The domain of the join attribute is from $[1, \frac{80{,}000{,}000}{K}]$ and each value appears exactly K times. For each K value, the stream contains 20 million tuples with a zipfian NoPerm distribution with skew 1 and the same domain as in the relation. Figure 7 depicts the throughput of each algorithm, when varying the size of the available memory. Again, SSIJ outperforms MeshJoin in all cases, supporting rates up to 3 times higher for 1:1 joins, and rates up to 9 times higher for 1:400 joins, with the performance benefits increasing with more available memory.

B. TPC-H Benchmark and Cloud Data Set

For the TPC-H experiment, the tables Part (which describes products) and LineItem (which contains order details) are used. This setting corresponds to a common scenario where an operational store ships information on new orders to a data warehouse. We also experimented with the Cloud data set. Performance of the joins is shown in Figure 8, as the available memory increases. Again, the benefits of SSIJ are very significant, especially for smaller memory sizes.

C. Comparison with Index Nested Loops

We now examine if the available out of the box relational join operators provided by relational engines are suitable for solving the problem of joining a stream with a relation. As the experiments in Figure 9 show, the answer is no. In this section we compare the service rate of SSIJ against the service rate of an index-based join algorithm provided by a leading open source (DBMS A) and by a leading commercial (DBMS B) database system. In this setup both the stream and the relation have an uniform distribution in the interval [0,250000]. The relational database system manages the stream as an in-memory relation with the size equal to 1 million tuples. The relation is stored on disk and it is indexed using a $B^{+}Tree$. The buffer pool size allocated to the database system is equal to 1GB. For configuring the database engine we ensured the system uses all relevant optimizations: the stream is stored in memory, the index is included in the query plan, the statistics are accurate and there is no transaction overhead. We measure the time to perform the index join between the stream and the relation and we compute the number of joined stream tuples per second. In this experiment we vary the size of the relation for all algorithms. The memory budget allocated to SSIJ 's operation is set to 10% of the relation size, and ranges from 50MB to 150MB. Our findings in Figure 9 show that SSIJ performs *at least* 3 times (and up to 24 times) faster than a join operator provided by a DBMS, even though the latter uses from 6 up to 20 times more memory buffers for the join and has the full stream loaded in memory in advance.

D. Sensitivity Analysis

We now evaluate the performance of SSIJ when we vary its parameters and the experiments presented in this section indicate the optimal settings for SSIJ. Additional experiments (not included due to the lack of space) show the range of values for which the algorithm has optimal performance to be robust across different data distributions. On the other hand,

for different relation sizes the shape of the graphs (for SB_{thres} and IB_{thres}, Figure 13 and Figure 12) remains the same but the values on the X axes are shifted left or right.

Dynamic Allocation Impact. memory budget of SSIJ is allocated to relation pages (including the $B^{+}tree$ relation index nodes) and the stream tuples, as described in Section IV-A. We now investigate the impact of this allocation. We measure the throughput of non-adaptive variants of SSIJ that use several different fixed allocation ratios to divide the memory between the relation and the stream, including a variant (No-Cache) where only the internal nodes of the $B^{+}tree$ index are cached. We compare them to each other and to SSIJ, which dynamically allocates the memory budget during its execution.

The trade-off of memory allocation in this context is the following: Allocating space to the relation saves disk I/O operations, which leads to faster processing of the stream tuples. At the same time, less space is allocated to the input buffer, which may not be able to accommodate incoming tuples while the algorithm is in its join phase.

In our experiment, whose results are shown in Figure 10, we focus on the case when the stream exhibits temporal locality, as described previously. In such an environment, when the range of values of the stream tuples changes, existing cache contents are less useful, leading to more reads from disk and potentially the need for greater input buffering. Thus, a fixed allocation scheme fails to keep up with the incoming tuples while a dynamic scheme allows SSIJ to evict these (less useful) cache pages making room for the input buffer. Once the quality of the cache content is improved the stream requires fewer I/O operations, which leads to a decreasing size of the IB and more memory space allocated to the CR.

Our findings can be summarized as follows: (i) Allocating memory for caching disk pages increases the performance of the algorithm. When 20%, 50% and 70% of the memory is allocated to the relation cache, the algorithm can support increasing throughput, and always higher compared to the case when no relation blocks are cached; (ii) The algorithm supports a significantly higher stream arrival rate when the memory budget is dynamically allocated between the relation and the stream. Moreover, the impact of the adaptive memory allocation increases as the memory size increases.

Another benefit of adaptive memory allocation is that it allows SSIJ to support bursts of arrival rates significantly higher than the nominal maximum supported rate, for nontrivial lengths of time. Specifically, let us focus on the previous setup, when the adaptive algorithm works in "steady state" (i.e., after the cache has been warmed up for each stream interval, so that the algorithm is exhibiting localized disk I/Os). Table III shows the maximum duration of a "burst" of arrivals that the algorithm can support when the burst rate is from 10% up to 70% *higher* than the nominal maximum supported rate. This experiment corresponds to Figure 10 where the nominal rate for this setting is equal to 480,000 tuples per second. Given a fixed memory budget, SSIJ is the only algorithm that supports bursts of arrivals higher then the nominal maximum supported rate; Mesh Join requires an increase in the allocated memory [5], [4].

Burst Rate Increase (% of max rate)	10%	25%	50%	70%
Burst Duration (ms)	700	562	400	225

TABLE III
DURATION OF BURSTS

Cache Replacement Policy. Figure 11 shows the impact of the cache replacement policy used by our algorithm for the same setup as the one in Figure 5, where the stream follows a zipf distribution with skew 1. We compare our caching policy with the Least Recently Used (LRU) technique. LRU considers blocks that are read during the join phase as more recent than the cached relation blocks, thus evicting useful relation blocks, which leads to inferior performance.

Sensitivity to the IB_{thresh} Parameter. We set the data distributions of the relation and the stream as in Figure 10. We allocate a memory budget equal to 10% of the size of the relation and we vary the size of the online phase threshold parameter IB_{thresh}. Figure 12 shows that increasing the number of stream tuples that are batched during the processes of scanning the index up to a point can significantly improve the performance of our algorithm.

Sensitivity to SB_{thresh} Parameter. The SB_{thresh} parameter influences the performance of our algorithm as shown in Figure 13. The stream follows a zipf distribution with skew value 1. The memory allocated for this experiment is equal to 1% of the relation size. All other parameters are set as in Table II. By increasing SB_{thresh}, the cost of fetching the disk blocks during the join phase is amortized among a larger number of stream tuples, leading to improved performance. For a large range of values, this cost remains almost constant. However, when the size of the buffer becomes a significant portion of the memory budget, then the performance of the algorithm is adversely affected. This happens not only because the join phase is delayed to such an extent that new incoming stream tuples in the input buffer cannot fit in memory (even after evicting the cache contents), but also because fewer pages fit in the cache.

Sensitivity to Filling Factor. In order to accommodate updates, SSIJ relation blocks have a relatively low *filling factor*. In this experiment we demonstrate the effect that the filling factor of leaf nodes in our used data structure has on the throughput of our algorithm. In Figure 14 we plot the results when the stream relation key values follow a zipfian NoPerm distribution with skew of 1. The default value of the filling factor in all the other experiments was set to a modest value of 0.8. We can see that the value of the filling factor may have a large impact on the throughput

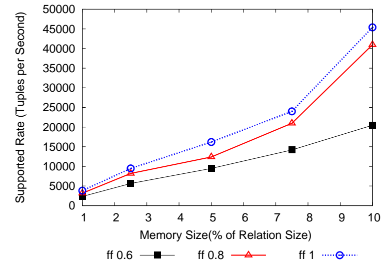

Fig. 14. Skew 1, ff = filling factor

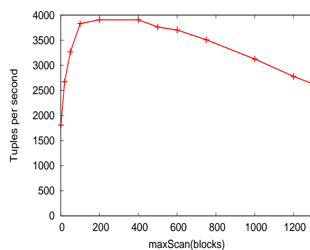

Fig. 15. Sensitivity to MaxScan

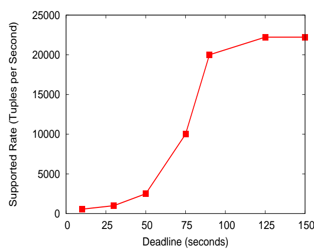

Fig. 16. Processing Deadlines

of our algorithm. This is not unexpected, as a smaller filling factor means that fewer tuples fit within a disk block, with an immediate effect on the caching benefit of each disk page as well as on the number of blocks that need to be read to satisfy matching requests by stream tuples.

Sensitivity to MaxScan. In Figure 15 we show the effect of $maxScan$ parameter. The stream follows a zipf distribution with skew value 1. The memory allocated for this experiment is equal to 1% of the relation size. All other parameters are set as in Table II. As expected, the algorithm has poor performance for small values of $maxScan$ since disk blocks are fetched without exploiting the opportunities for sequential scans. However, very large sizes of $maxScan$ may lead to reads that evict a large number of useful relation pages.

Processing Deadlines. In Figure 16 we investigate how the maximum supported rate of SSIJ varies when the algorithm is asked to process each stream tuple within a given deadline. In this experiment the size of the relation is equal to 5GB and the join attribute follows an uniform distribution. The stream follows a zipf distribution with skew 1. The memory budget allocated to SSIJ in equal to 7.5% of the relation size.

The rate supported by SSIJ is equal to 550 tuples per second when each tuple is processed within a 10 seconds deadline. As we increase the deadline value up to 2 minutes the supported rate also increases, up to a rate of 22000 tuples per second. In the presence of tight deadlines, SB_{thres} is never reached and the join phase is triggered by the stream tuples' expiration time. For tight deadlines the effect is the same as having a small SB_{thres} value which, as explained in the experiment of Figure 13, decreases the supported rate. This is significantly better than existing alternatives. In particular, for the same setup MeshJoin gives tuples delays of more than 9 minutes while supporting a rate of only 5500 tuples per second. For comparison, for a rate of 5500 tuples per second, SSIJ can keep deadlines of 1 minute. Additionally, the work in [6] which specifically targets reducing latency can result in latencies even longer than MeshJoin.

VII. Conclusions

We investigate the problem of efficiently joining a streamed relation with a stored one. The problem is motivated by the ETL processes of active data warehouses and also by applications of data streams and by the need to combine sensor and historical data. We propose SSIJ, a new semi-streaming exact join algorithm that supports very fast streams and exploits very well the (small or large) available memory. The operator easily supports pipelined execution plans, which is important in many DSMS applications [20] and online ETL processes. Our experiments demonstrate that SSIJ consistently outperforms existing techniques in terms of the maximum sustainable throughput of the join, for a variety of synthetic and real data sets. The benefits of the proposed algorithm increase with increased levels of skew, which is typically observed in real data sets. Future work includes extending the algorithm's applicability to even faster streams via principled load shedding for result approximation.

ACKNOWLEDGMENTS: We would like to thank the authors of [4] for making their code available to us.

References

[1] R. Kimball and J. Caserta, *The Data Warehouse ETL Toolkit*. Wiley Publishing, Inc., 2004.
[2] T. Barclay, R. Barnes, J. Gray, and P. Sundaresan, "Loading Databases Using Dataflow Parallelism," *SIGMOD Record*, vol. 23, no. 4, 1994.
[3] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos, "Cubetree: Organization of and Bulk Incremental Updates on the Data Cube," in *ACM SIGMOD*, 1997, pp. 89–99.
[4] N. Polyzotis, S. Skiadopoulos, P. Vassiliadis, A. Simitsis, and N.-E. Frantzell, "Supporting streaming updates in an active data warehouse," *ICDE*, pp. 476–485, 2007.
[5] N. Polyzotis et al., "Meshing streaming updates with persistent data in an active data warehouse," *IEEE TKDE*, vol. 20, no. 7, pp. 976–991, 2008.
[6] A. Chakraborty and A. Singh, "A partition-based approach to support streaming updates over persistent data in an active data warehouse," *IEEE Intern. Symposium on Parallel and Distributed Processing*, 2009.
[7] D. Burleson, "New Developments in Oracle Data Warehousing," 2004.
[8] O. Corp., "On-time data warehousing with oracle10g - information at the speed of your business," August 2003.
[9] R. J. Santos and J. Bernardino, "Real-time data warehouse loading methodology," in *IDEAS*, 2008, pp. 49–58.
[10] C. White, "Intelligent Business Strategies: Real-time Data Warehousing Heats up," *DM Review*, 2002.
[11] W. Labio and H. Garcia-Molina, "Efficient Snapshot Differential Algorithms for Data Warehousing," in *VLDB*, 1996, pp. 63–74.
[12] W. Labio, J. Wiener, H. Garcia-Molina, and V. Gorelik, "Efficient Resumption of Interrupted Warehouse Loads," in *ACM SIGMOD*, 2000.
[13] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom, "Performance Issues in Incremental Warehouse Maintenance," in *VLDB*, 2000.
[14] J. M. Cheng, D. J. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, and Y. Wang, "An efficient hybrid join algorithm: A db2 prototype," in *ICDE*, 1991.
[15] R. P. Kooi, "The optimization of queries in relational databases," Ph.D. dissertation, 1980.
[16] M. Elhemali et al., "Execution strategies for sql subqueries," in *SIGMOD '07*.
[17] B. Seeger, P.-A. Larson, and R. McFayden, "Reading a set of disk pages," in *VLDB*, 1993.
[18] S. Chandrasekaran et al., "Telegraphcq: Continuous Dataflow Processing," in *ACM SIGMOD*, 2003.
[19] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," in *SIGMOD*, 2003.
[20] D. Carney et al., "Monitoring Streams: A New Class of Data Management Applications," in *VLDB*, 2002.
[21] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Rec.*, vol. 30, no. 3, pp. 109–120, 2001.
[22] S. Chandrasekaran and M. Franklin, "Remembrance of streams past:," in *VLDB*. VLDB, 2004, pp. 348–359.
[23] R. Bayer and E. M. McCreight, "Organization and Maintenance of Large Ordered Indices," *Acta Inf.*, vol. 1, pp. 173–189, 1972.
[24] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes," in *ACM SIGMOD*, 1997, pp. 38–49.
[25] T. Johnson and D. Shasha, "The performance of current b-tree algorithms," *ACM Trans. Database Syst.*, vol. 18, no. 1, 1993.
[26] N. Tatbul, "Streaming data integration: Challenges and opportunities," in *NTII*, 2010.