An Efficient Stream-based Join to Process End User Transactions in Real-Time Data Warehousing

M. Asif Naeem[1], Noreen Jamil[2]
[1]School of Computing and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand

[2]Department of Computer Science
The University of Auckland
Auckland, New Zealand
mnaeem@aut.ac.nz, njam031@aucklanduni.ac.nz

Journal of Digital Information Management

ABSTRACT: *In the field of real-time data warehousing semistream processing has become a potential area of research since last one decade. One important operation in semi-stream processing is to join stream data with a slowly changing diskbased master data. A join operator is usually required to implement this operation. This join operator typically works under limited main memory and this memory is generally not large enough to hold the whole disk-based master data. Recently, a seminal join algorithm called MESHJOIN (Mesh Join) has been proposed in the literature to process semistream data. MESHJOIN is a candidate for a resource-aware system setup. However, MESHJOIN is not very selective. In particular, MESHJOIN does not consider the characteristics of stream data and its performance is suboptimal for skewed stream data. In this paper we propose a novel Semi-Stream Join (SSJ) using a new cache module. The algorithm is more appropriate for skewed distributions, and we present results for Zipfian distributions of the type that appears in many applications. We present the cost model for our SSJ and validate it with experiments. Based on the cost model we also tune the algorithm up to a maximum performance. We conduct a rigorous experimental study to test our algorithm. Our experiments show that SSJ outperforms MESHJOIN* significantly.

Subject Categories and Descriptors:

H.2.7 [Database Administration]; Data Warehouse and Repository: **H.2.4 [Systems];** Transaction Processing

General Terms: Data Warehousing, Data Processing

Keywords: Real-time Data Warehousing, Semi-stream Processing, Join Operator, Performance Measurement

Received: 3 January 2014, Revised 13 January 2014, Accepted 9 February 2014

1. Introduction

In the field of Data Stream Management (DSM), stream processing due to its infinite characteristics has become a potential area of research over the last decade. Data stream processing deals with continuously arriving information, which is important for many different applications such as network traffic monitoring [1], sensor data [2], web log analysis [3], online auctions [4], and supply-chain management [5]. One kind of stream processing is to join single stream data with slowly changing disk-based data using a stream-based join operator. A typical example of such type of stream processing is in real-time data warehousing [6] [7]. In this application, the slowly changing data is typically a master data table while incoming real-time sales data (also called end user transactions) is a stream data. The stream based join can be used for example to replace data source key with warehouse key or enrich the stream data with master data. The most natural type of join in this scenario would be an equijoin, performed for example on a foreign key in the stream data.

In the literature, a well known semi-stream algorithm MESHJOIN [8] [9] was proposed for joining a continuous stream data with a disk-based master data, such as the scenario in active data warehouses. The MESHJOIN algorithm is a hash join, where the stream serves as the build input and the disk-based relation serves as the probe input.

The algorithm performs a staggered execution of the hash table build in order to load in stream tuples more steadily. Although the MESHJOIN algorithm efficiently amortizes the disk I/O cost over fast input streams, the algorithm makes no assumptions about characteristics of stream data or the organization of the master data. Experiments by the MESHJOIN authors have shown that the algorithm performs worse with skewed data. Therefore, the question remains how much potential for improvement remains untapped due to the algorithm not being consider the characteristics of stream data.

In this paper we focus on one of the most common characteristics, a skewed distribution. Such distributions arise in practice, for example current economic models show that in many markets a selective few products are bought with higher frequency [10]. Therefore, in the input stream, the end user transactions related to those products are the most frequent. In MESHJOIN, the algorithm does not consider the frequency of stream tuples.

We propose a robust algorithm called Semi-Stream Join (SSJ). The key feature of SSJ is that the algorithm stores the most used portion of the disk-based relation, which matches the frequent items in the stream, in memory. As a result, this reduces the I/O cost substantially, which improves the performance of the algorithm. Since our purpose is primarily to gauge performance with skewed distributions, we consider a very clean, artificial as well as real datasets that exactly exhibit a well-understood type of skew, a power law.

The rest of the paper is structured as follows. Section II presents related work. The existing MESHJOIN and problem statement are defined in Section III. Section IV describes the proposed SSJ with its execution architecture and cost model. Section V-A presents the extension of SSJ in the form of tuning. Section VI describes an experimental study of SSJ. Finally, Section VII concludes the paper.

2. Related Work

In this section we will outline the well known work that has already been done in this area with a particular focus on those which are closely related to our problem domain.

The non-blocking symmetric hash join (SHJ) [11] promotes the proprietary hash join algorithm by generating the join output in a pipeline. In the symmetric hash join there is a separate hash table for each input relation. When the tuple of one input arrives it probes the hash table of the other input, generates a result and stores it in its own hash table. SHJ can produce a result before reading either input relation entirely, however, the algorithm keeps both the hash tables, required for each input, in memory. Early Hash Join (EHJ) [12] is a further extension of SHJ.

The Double Pipelined Hash Join (DPHJ) [13] with a two stage join algorithm is an extension of SHJ. The XJoin algorithm [14] is another extension of SHJ. Hash-Merge Join (HMJ) [15] is also one based on symmetric join algorithm. It is based on push technology and consists of two phases, hashing and merging.

Early Hash Join (EHJ) [12] is a further extension of SHJ. EHJ introduces a new biased flushing policy that flushes the partitions of the largest input first. EHJ also simplifies the strategies to determine the duplicate tuples, based on cardinality and therefore no timestamps are required for arrival and departure of input tuples. However, because EHJ is based on pull technology, a reading policy is required for inputs.

R-MESHJOIN (reduced Mesh Join) [16] clarifies the dependencies among the components of MESHJOIN. As a result, it improves the performance slightly. However, R-MESHJOIN again does not consider the non-uniform characteristic of stream data.

One approach to improve MESHJOIN is a partitionbased join algorithm [17] that can also deal with stream intermittence. It uses a two-level hash table for attempting to join stream tuples as soon as they arrive, and uses a partition-based waiting area for other stream tuples. For the algorithm in [17], however, the time that a tuple is waiting for execution is not bounded. We are interested in a join approach where there is a time guarantee for when a stream tuple will be joined.

Another recent approach, Semi-Streaming Index Join (SSIJ) [18] joins stream data with disk-based data. SSIJ uses page level cache i.e. stores the entire disk pages in cache while it is possible that all the tuples in these pages may not be frequent in stream. As a result the algorithm can perform suboptimal. Also the algorithm does not include the mathematical cost model.

3. Preliminaries and Problem Definition

In this section we summarize the MESHJOIN algorithm and at the end of the section we describe the observations that we focus on in this paper.

MESHJOIN was designed to process stream data (also called end user transactions) with disk-based master data (also called disk-based relation) in the field of real time data warehousing. The algorithm reads the disk-based relation R sequentially in segments. Once the last segment is read, it again starts from the first segment. The algorithm contains a buffer, called the disk buffer, to store each

segment in memory one at a time, and has a number of memory partitions, equal in size, to store the stream tuples. These memory partitions behave like a queue and are differentiated with respect to the loading time. The number of partitions is equal to the number of segments on the disk while the size of each segment on the disk is equal to the size of the disk buffer. In each iteration the algorithm reads one disk segment into the disk buffer and loads a chunk of stream tuples into the memory partition. After loading the disk segment into memory it joins each tuple from that segment with all stream tuples available in different partitions. Before the next iteration the oldest stream tuples are expired from the join memory and all chunks of the stream are advanced by one step. In the next iteration the algorithm replaces the current disk segment with the next one, loads a chunk of stream tuples into the memory partition, and repeats the above procedure.

The MESHJOIN algorithm successfully amortizes the fast arrival rate of the incoming stream by executing the join of disk pages with a large number of stream tuples. However there are still some further issues that exist in the algorithm. MESHJOIN does not consider the characteristic of skew in stream data. Experiments by the MESHJOIN authors have shown that the algorithm performs suboptimal with skewed data.

4. Semi-stream Join (SSJ)

In this paper, we propose a new algorithm, Semi-Stream Join (SSJ), that overcomes the issues stated in above section. This section gives a detail overview of the SSJ algorithm and presents its cost model.

4.1 Execution Architecture

The SSJ algorithm possesses two complementary hash join phases, somewhat similar to Symmetric Hash Join. One phase uses R as the probe input; the largest part of R will be stored in tertiary memory. We call it the disk-probing phase. The other join phase uses the stream as the probe input, but will deal only with a small part of relation R. We call it stream-probing phase. For each incoming stream tuple, SSJ first uses the stream-probing phase to find a match for frequent requests quickly, and if no match is found, the stream tuple is forwarded to the disk-probing phase.

The execution architecture for SSJ is shown in Figure 1. The largest components of SSJ with respect to memory size is hash table H_S that stores stream tuples. The other main components of SSJ are a disk buffer, a queue, a stream buffer, and another hash table H_R. Hash table H_R, for R contains the most frequently accessed part of R and

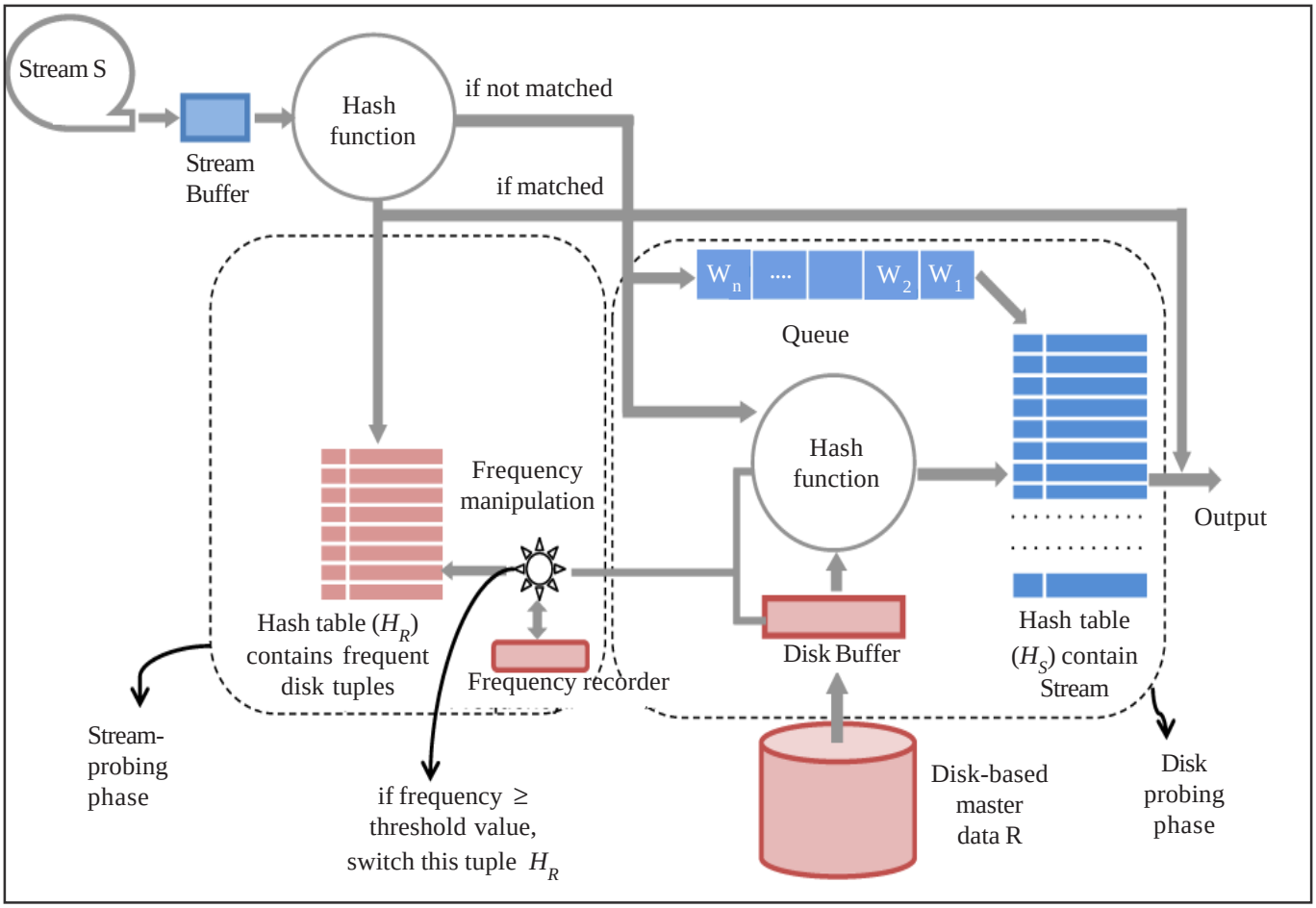


Figure 1. Execution architecture of SSJ

is stored permanently in memory. Relation R and stream S are the external input sources.

SSJ alternates between the stream-probing and the diskprobing phases. The hash table H_S is used to store only that part of the update stream that does not match tuples in H_R. A stream-probing phase ends if H_S is completely filled or if the stream buffer is empty. Then the disk-probing phase becomes active. In each iteration of the disk-probing phase, the algorithm loads a set of tuples of R into memory to amortize the costly disk access. After loading the disk pages into the disk buffer, the algorithm probes each tuple of the disk buffer in the hash table H_S. If the required tuple is found in H_S, the algorithm generates that tuple as an output. After each iteration the algorithm removes the oldest chunk of stream tuples from H_S. This chunk is found at the top of the queue; its tuples were joined with the whole of R and are thus completely processed now.

As the algorithm reads R sequentially, no index on R is required. After one iteration of disk-probing phase, a sufficient number of stream tuples are deleted from H_S, so the algorithm switches back to the stream-probing

Lines 2 to 9 specify the stream-probing phase. In this phase the algorithm reads w stream tuples from the stream buffer (line 2). After that the algorithm probes each tuple t of w in the disk-build hash table H_R, using an inner loop phase. One phase of stream-probing with a subsequent phase of disk-probing constitutes one outer iteration of SSJ.

The stream-probing phase (also called cache module) is used to boost the performance of the algorithm by quickly matching the most frequent master data. For determining very frequent tuples in R and loading them into H_R, the frequency detection process is required. This process tests whether the matching frequency of the current tuple is larger than a pre-set threshold. If it is, then this tuple is entered into H_R. If there are no empty slots in H_R the algorithm overwrites an existing least frequent tuple in H_R. This least frequent tuple is determined by the component frequency recorder.

An important question is how frequently a master data tuple must be used in order to get into this phase, so that the memory sacrificed for this phase really delivers a performance advantage. In Section V-A we give a precise

Algorithm 1 SSJ

Input: A disk based relation R and a stream of updates S

Output: $R \bowtie S$

Parameters: w (where $w = w_S + w_N$) tuples of S and b tuples of R.

Method:

```
1:  while (true) do
2:    READ w stream tuples from the stream buffer
3:    for each tuple t in w do
4:      if t ∈ H_R then
5:        OUTPUT t
6:      else
7:          ADD stream tuple t into H_S and also place its
            pointer value into Q
8:      end if
9:    end for
10:   READ b number of tuples of R into the disk buffer
11:   for each tuple r in b do
12:     if r ∈ H_S then
13:       OUTPUT r
14:       f ← number of matching tuples found in H_S
15:       if (f ≥ thresholdValue) then
16:         SWITCH the tuple r into hash table H_R
17:       end if
18:       end if
19:     end for
20:     DELETE the oldest w tuples from H_S along with
        their corresponding pointers from Q
21:   end while
```

and comprehensive analysis that shows that a remarkably small amount of memory assigned to the stream-probing phase can deliver a substantial performance gain.

4.2 Algorithm

The execution steps for SSJ are shown in Algorithm 1. The outer loop of the algorithm is an endless loop, which is common in stream processing algorithms (line 1). The body of the outer loop has two main parts: the stream-probing phase and the disk-probing phase. Due to the endless loop, these two phases alternate.

Lines 2 to 9 specify the stream-probing phase. In this phase the algorithm reads w stream tuples from the stream buffer (line 2). After that the algorithm probes each tuple t of w in the disk-build hash table H_R, using an inner loop (line 3). In the case of a match, the algorithm generates the join output without storing t in H_S. In the case where t does not match, the algorithm loads t into H_S, while also enqueuing its pointer in the queue Q (lines 4-8).

Lines 10 to 20 specify the disk-probing phase. At the start of this phase, the algorithm reads b tuples from R and loads them into the disk buffer (line 10). In an inner loop, the algorithm looks up all tuples from the disk buffer in hash table H_S. In the case of a match, the algorithm generates that tuple as an output (lines 11 to 13). Since H_S is a multihash-map, there can be more than one match; the number of matches is f (line 14).

Lines 15 and 16 are concerned with frequency detection. In line 15 the algorithm tests whether the matching frequency f of the current tuple is larger than a pre-set threshold. If it is, then this tuple is entered into H_R. If there are no empty slots in H_R, the algorithm overwrites an existing least-frequent tuple in H_R using the frequency recorder. Finally, the algorithm removes the expired stream tuples (i.e. the ones that have been joined with the whole of R) from H_S, along with their *pointer* values from the queue (line 20). If the cache is not full, this means the threshold is too high; in this case, the threshold can be lowered automatically. Similarly, the threshold can be raised if tuples are evicted from the cache too frequently. This makes the stream-probing phase flexible and able to adapt online to changes in the stream behavior. Necessarily, it will take some time to adapt to changes, similar to the warmup phase. However, this is usually deemed acceptable for a stream-based join that is supposed to run for a long time.

4.3 Cost model

In this section we develop the cost model for our proposed SSJ. The main objective for developing our cost model is

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

Table 1. Notations used in cost estimation of SSJ

to interrelate the key parameters of the algorithm, such as input size w, processing cost c_{loop} for these w tuples, the available memory M and the service rate μ. The cost model presented here follows the style used for MESHJOIN [9] [8]. Equation 1 represents the total memory used by the algorithm (except the stream buffer), and Equation 2 describes the processing cost for each iteration of the algorithm. The notations we used in our cost model are given in Table 1.

4.3.1 Memory cost

The major portion of the total memory is assigned to the hash table H_S together with the queue while a comparatively much smaller portion is assigned to H_R and the disk buffer. The memory for each component can be calculated as follows:

Memory for disk buffer (bytes) = $k.vP$

Memory for H_R (bytes) = $l.vP$

Memory for frequency recorder (bytes) = $8h_R$

Memory for H_S (bytes) = $[M - (k + l)\ v_P\ 8h_R]$

Memory for the queue (*bytes*) = $(1 - \alpha)\ [M - (k + l)\ v_P 8h_R]$

By aggregating the above, the total memory for SSJ can be calculated as shown in Equation 1.

$$M = (k + l)\ v_P + 8h_R + \alpha\,[M - (k + l)\ v_P - 8h_R] + (1 - \alpha)\,[M\ (k + l)\ v_P - 8h_R] \quad (1)$$

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

4.3.2 Processing cost

In this section we calculate the processing cost for the algorithm. To make it simple we first calculate the processing cost for individual components and then sum these costs to calculate the total processing cost for one iteration.

$c_{I/O}(k\,.v_P)$ = Cost to read k pages into the disk buffer

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

$d.c_H$ = Cost to look-up disk buffer tuples in H_S

$d.c_F$ = Cost to compare the frequency of all the tuples in disk buffer with the threshold value

$w_N.c_O$ = Cost to generate the output for w_N tuples

$w_S.c_O$ = Cost to generate the output for w_S tuples

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

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

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

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

By aggregating the above costs the total cost of the algorithm for one iteration can be calculated using Equation 2.

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

The term 10^{-9} is a unit conversion from nanoseconds to seconds. In c_{loop} seconds the algorithm processes w_N and w_S tuples of the stream S, the service rate μ can be calculated using Equation 3.

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

5. Extensions

This section presents the tuning of SSJ as an our extended work.

5.1 Tuning

As we have outlined in the abstract, we assume that only limited resources are available for SSJ. Hence we face a trade-off with respect to memory distribution. Assigning more memory to one component means assigning equally less memory to some other components. Therefore, to utilize the available memory optimally, tuning of the join components is important. If the size of R and the overall memory size M is fixed, the equation is a function of two parameters, the size for disk buffer and the size of hash table H_R.

The tuning of the algorithm uses the cost model that we have derived. Therefore we decided to use the tuning of the algorithm to experimentally validate the cost model. We not only provide a theoretical approach to tuning, based on calculus of variations. We first approximate optimal tuning settings using an empirical approach, by considering a sample of values for the disk buffer and hash table H_R. Finally we compare the experimentally obtained tuning results with the results obtained based on the cost model.

5.1.1 Empirical Tuning

This section focuses on obtaining samples for the approximate tuning of the key components. Since, the performance is a function of two variables, the size of the disk buffer, d, and the size of hash table H_R, h_R. We tested the performance of the algorithm for a grid of values for both components, i.e. for each setting of d the performance is measured against a series of values for h_R. The performance measurements for the grid of d and h_R are shown in Figure 2. It is worth following the data along the h_R axis, i.e. for a fixed d we look at all values for h_R. This will show that a stream-probing phase is useful if it remains within a certain size. This is so, because in the beginning the performance increases rapidly with an increase in h_R. Then, after reaching an optimum the performance decreases. The explanation is that when h_R is increased beyond this value, it does not make any significant

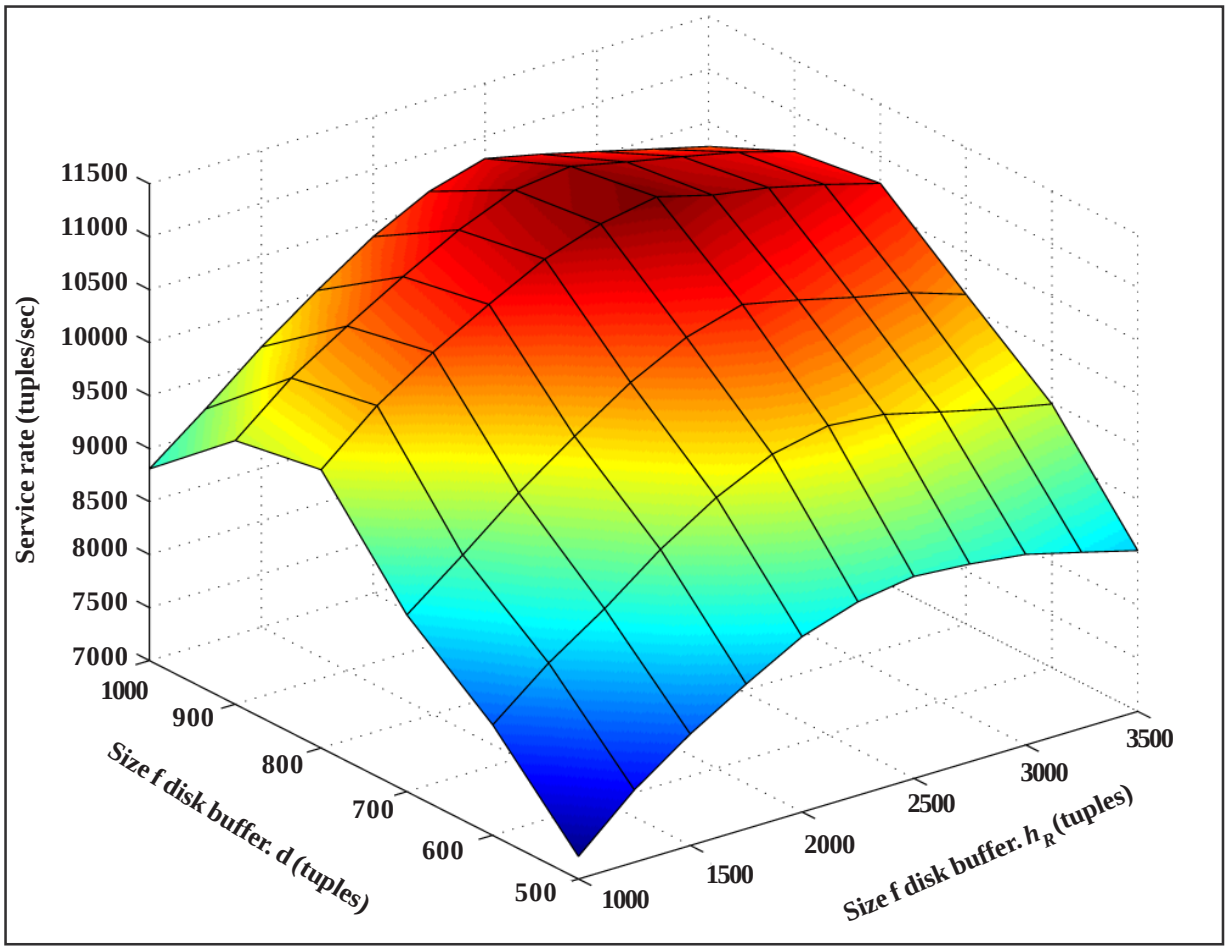


Figure 2. Tuning of SSJ using measurement approach

difference to the stream matching probability due to the characteristics of the skewed distribution. On the other hand it reduces the memory for hash table H_S. Similarly we can follow the data along the d axis. Initially the performance increases, since the costly disk access is amortized for a larger number of stream tuples. This effect is actually of crucial importance, because it is this gain that gives the algorithm an advantage over a simple index-based join. It is here that H_S is used in order to match more tuples than just the one that was used in order to determine the partition that was loaded. After attaining a maximum, the performance decreases because of the increase in I/O cost for loading more of R at one time in a non-selective way.

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

5.1.2 Tuning based on cost model

We now show how the cost model for SSJ can be used to (theoretically) obtain an optimal tuning of the components. Equation 1 and 2 represents the memory and processing cost respectively for the algorithm. On the basis of these equations the performance of the algorithm can be calculated using Equation 3.

The algorithm can be tuned to perform optimally using Equation 3 by knowing w_N, w_N and c_{loop}. The value of cloop can be calculated from Equation 2 if we know w_N and w_S.

Mathematical model for w_N: SSJ has two separate phases, the stream-probing and the disk-probing phase. The stream tuples that are matched in the stream-probing phase are joined straight away without storing them in H_S. The number of tuples processed through this phase per outer iteration are denoted by w_N.

The main components that directly affect w_N are the size of the master data on disk and the size of H_R. To calculate the effect of both components on w_N we assume that R_t is the total number of tuples in R while h_R is the size of H_R in terms of tuples. We now use our assumption that the stream of updates S has a Zipfian distribution with exponent value one. In this case the matching probability for S in the stream-probing phase can be determined using Equation 4. The denominator is a normalization term to ensure all probabilities sum up to 1.

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

Each summation in the above equation generates the harmonic series which can be summed up using formula $\sum_{x=1}^{k} \frac{1}{x} = \ln k + \gamma + \varepsilon_k$ [19], where γ is Euler's constant whose value is approximately equal to 0.5772156649 and ε_k is another constant which is $\approx \frac{1}{2k}$. The value of ε_k approaches 0 as *k* goes to ∞ [19]. In our case the value of $\frac{1}{2k}$ is small so we ignore it. Hence Equation 4 can be written as shown in Equation 5.

$$p_N = \frac{In\ h_R}{In\ R_t} \tag{5}$$

Now using Equation 5 we can determine the constant factors of change in p_N by changing the values of h_R and R_t individually. Let us assume that p_N decreases with constant factor ϕ_N by doubling the value of R_t and increase with constant factor ψ_N by doubling the value of h_R. Knowing these constant factors we are able to calculate the value of w_N. Let us assume the following:

$$p_N = R_t^{y}\ h_R^{z} \tag{6}$$

Dividing the above equation by Equation 6 we get $2_y = \phi_N$ and therefore, $y = \log_2(\phi_N)$:

Determination of *z*: Similarly we also know that by doubling h_R the matching probability p_N increases by a constant factor N therefore, Equation 6 can be written as:

$$\psi_{NpN} = (2R_t)^{y} h_R^{z})^2$$

By dividing the above equation by Equation 6 we get $2z = \psi_N$ and therefore, $2^z = log_2(\psi_N)$. After substituting the values of constants *y* and *z* into Equation 6 we get:

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

Now if S_n is the total number of stream tuples that are processed (through both phases) in n outer iterations then w_N can be calculated using Equation 7.

$$w_N = \frac{(R_t^{\log_2(\phi_N)}\ h_R^{\log_2(\psi_N)})S_n}{n} \tag{7}$$

Mathematical model for w_S: The second phase of the SSJ algorithm deals with the rest of *R*. This part is called *R*′, with $R' = R - h_R$. The algorithm reads *R*′ in segments. The size of each segment is equal to the size of the disk buffer d. In each iteration the algorithm reads one segment of *R*′ using an index on the join attribute and loads it into the disk buffer. Since we assume a skewed distribution, the matching probability is not equal, but decreases in the tail of the distribution, as shown in Figure 3. We calculate the matching probability for each segment by summing over the discrete Zipfian distribution separately and then aggregating all of them as shown below.

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

We simplify this to:

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

From this we can obtain the average matching probability $\overline{p}_S$ in the disk-probing phase, which we need for calculating w_S. Let N be the total number of segments in *R*′. In the denominator, we have to use the same normalization term as in Equation 4.

$$\overline{p}_S = \frac{\sum_{x=h_R+1}^{R_t} \frac{1}{x}}{N \sum_{x=1}^{R_t} \frac{1}{x}}$$

We again use the summation formula [19]:

$$\overline{p}_S = \frac{In\,(R_t) - In\,(h_R)}{N(In\,(R_t) + \gamma)} \tag{8}$$

To determine the effects of d, h_R and R_t on $\overline{p}_S$, a similar argument can be used as in the case of w_N. Let's suppose we double *d* in Equation 8, then *N* will be halved and the value of p_S increases by a constant factor of *S*. Similarly, if we double h_R or R_t respectively, then the value of p_S decreases by some constant factor of ψ_S or ϕ_S respectively. Using a similar argument for w_N, we get:

$$\overline{p}_S = d^{x} h_R^{y} R_t^{z} \tag{9}$$

The values for the constants *x*, *y* and *z* in this case will be $x = \log_2(\theta_S)$, $y = \log_2(\psi_S)$ and $z = \log_2(\phi_S)$ respectively. Therefore by replacing the values with constants Equation 9 will become.

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

Now if h_S are the number of stream tuples stored in the hash table then the average value for w_S can be calculated using Equation 10.

$$w_S(average) = d^{\log_2(\theta_S)}\ h_R^{\log_2(\psi_S)}\ R_t^{\log_2(\phi_S)} h_S \tag{10}$$

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

5.1.3 Comparison of cost model and measurements

The terms w_S and w_N are of crucial importance in our understanding of SSJ. This is so, firstly because they refer to the functioning of both join phases. Secondly they

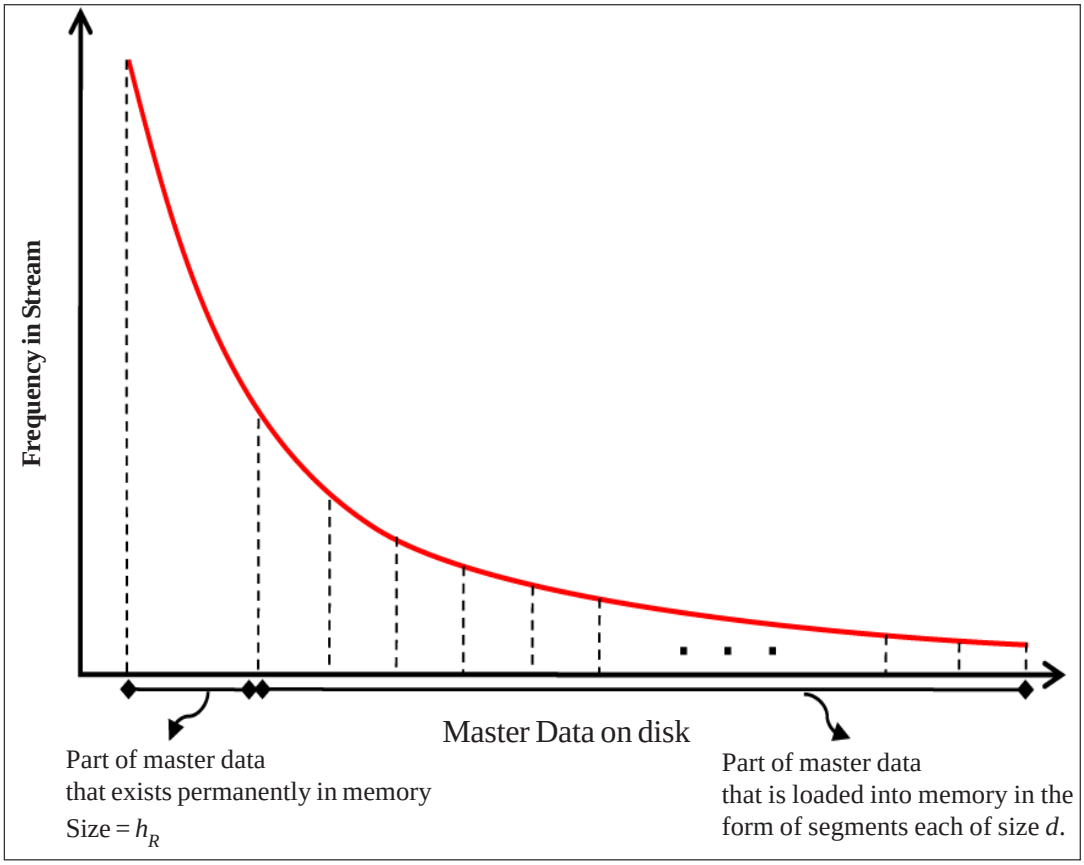


Figure 3. Matching probability of R in stream

are in a complex relationship: the success of one join iteration determines how many resources are available for the next step. This relationship is fully covered in our cost model presented in IV-C. That cost model was in turn empirically validated. For due diligence we feel that it is important to empirically validate the model predictions for w_S and w_N.

Therefore we now present cost model predictions i.e. calculated values for w_N and w_S, as well as the corresponding measurements of w_N and w_S from our experiments. Figures 4(a) to (f) contain graphs showing predicted and measured values for w_N and w_S while one system parameter is changed, i.e. all other parameters stay the same. Hence the figures show the effect of key system parameters on w_N and w_S. The parameters that are varied are named in the caption. For instance, in Figure 4 (a) and (b) the parameter is the size of H_R, while the memory is held constant. This means these figures refer to different possible settings before tuning. The tuning process will pick only one setting, namely the optimal setting. These figures are helpful in understanding the tuning process further. The service rate values used in Section V-A1 are influenced by these two values, as is clear from Equation 3. Due to the denominator, this is a bit more complex than just the sum of w_N and w_S, but just looking at the sum $w_N + w_S$ gives an approximate picture of the tuning challenge. The fact is that we face a trade-off. Increasing the size of H_R will increase the one parameter and decrease the other. The tuning process picks the golden mean. The measurements show good correspondence between predictions and measurements. Figures 4(c) to (f) are provided as further reference to understand the inner workings of the algorithm.

5.1.4 Comparison between tuning approaches

We can now compare the tuning results obtained through measurements with the tuning results that we calculated using the cost model. Figure 5 (a) shows the empirical and the mathematical tuning results for the disk buffer size d. One can say that the results in both cases are reasonably similar, with a deviation of only 2.5%.

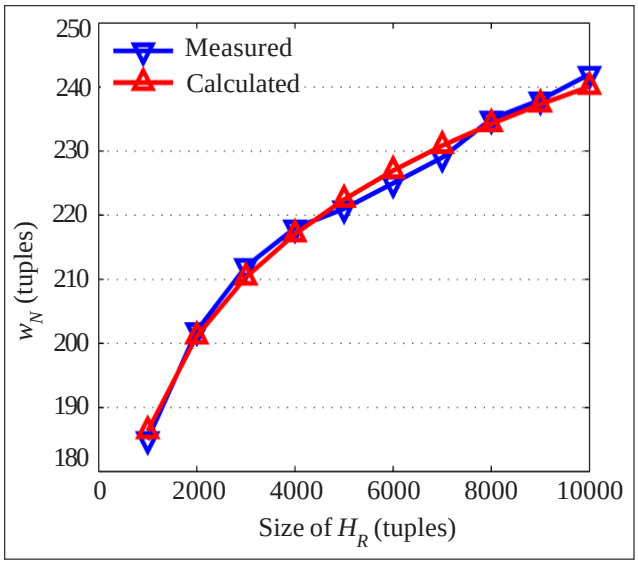
Figure 5 (b) shows the empirical and the mathematical tuning results for the hash table size H_R. Again we think it is fair to say that the results in both cases are reasonably similar, with a deviation of only 0.65%. This is a corroboration of the accuracy of our cost model.

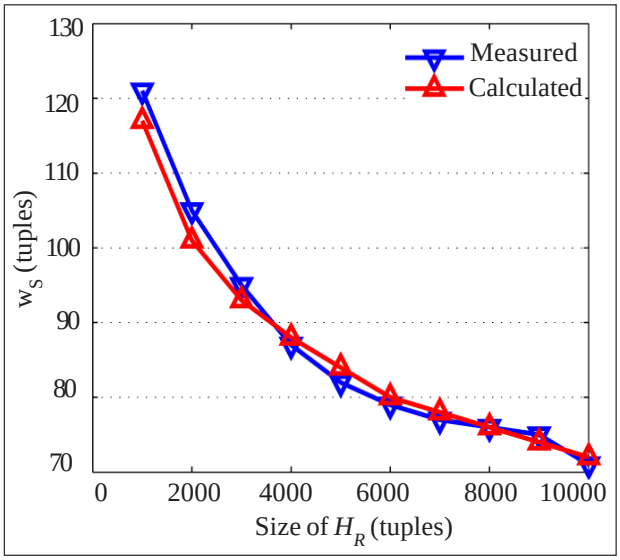
6. Performance Experiments

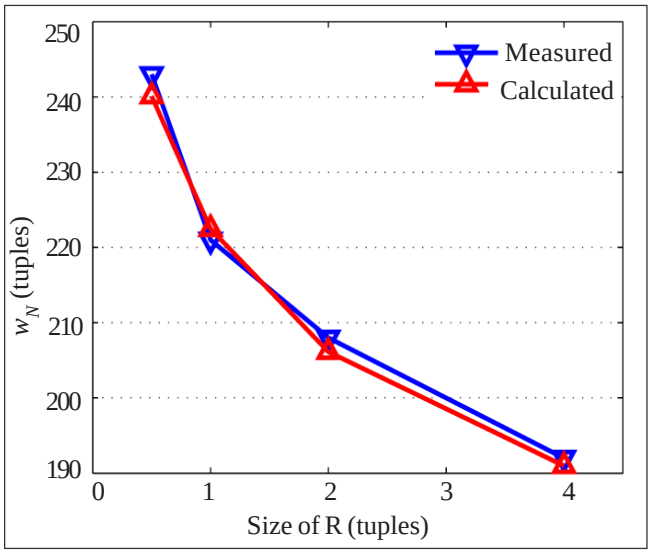
6.1 Experimental Setup

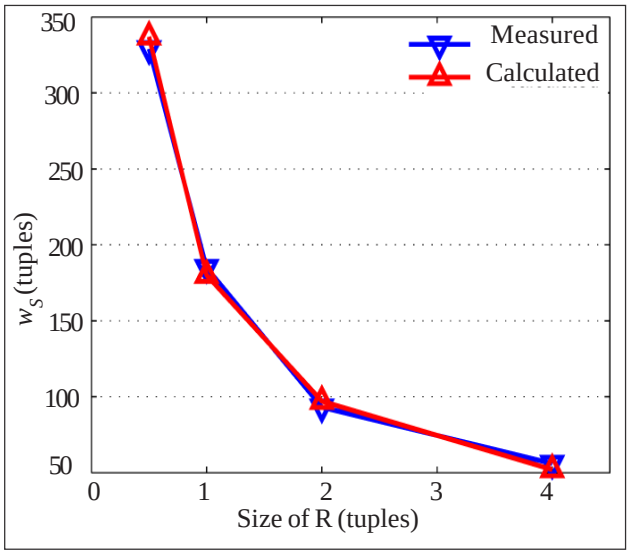
Hardware specification: We performed our experiments on a Pentium-core-i5 with 8GB main memory and 500GB hard drive as a secondary storage. We implemented our experiments in $Java$ using the $Eclipse\ IDE$. The relation R is stored on disk using a $MySQL$ database.

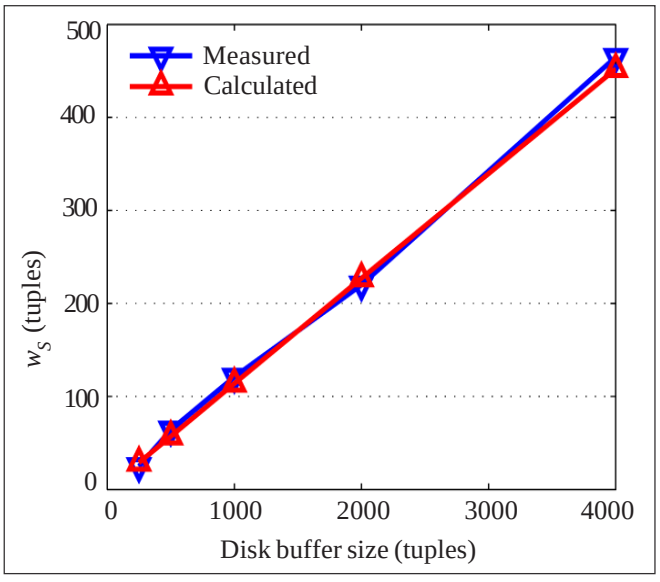
Measurement strategy: The performance or service rate











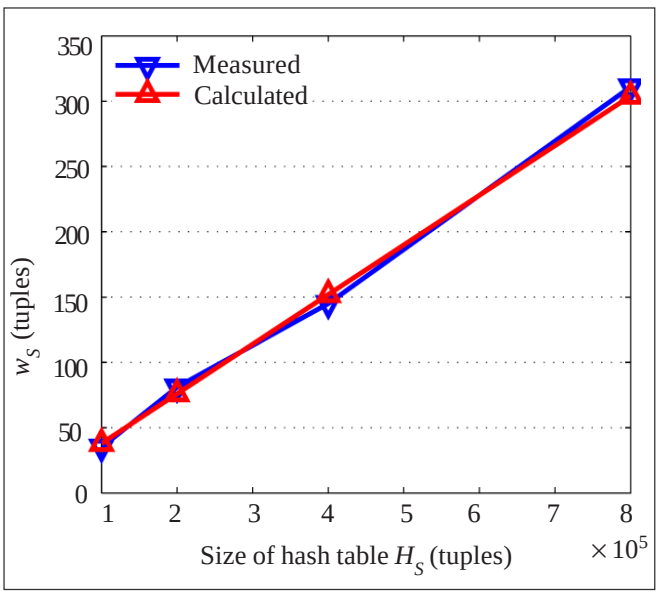
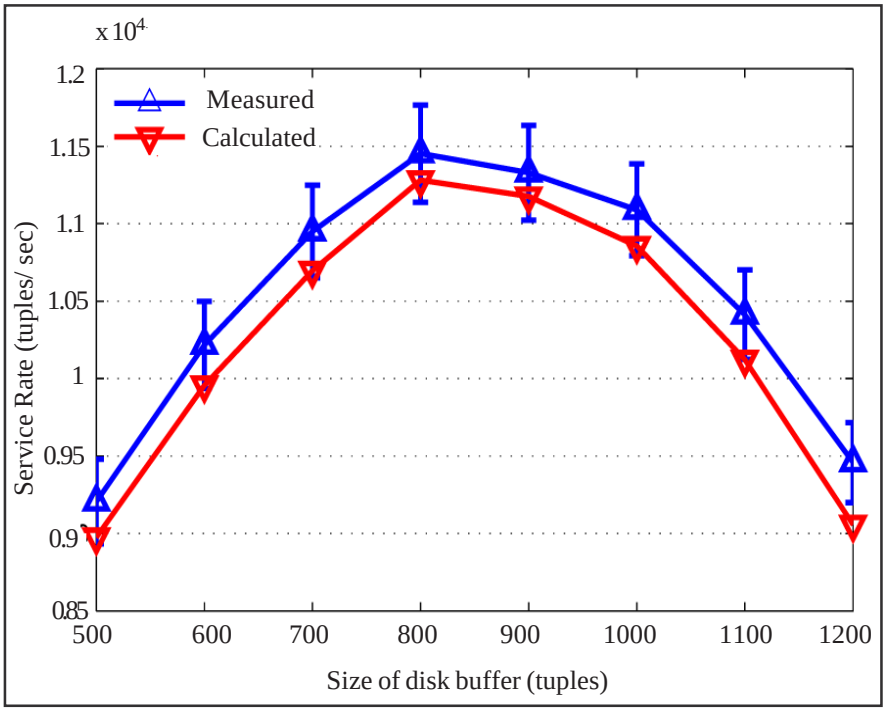
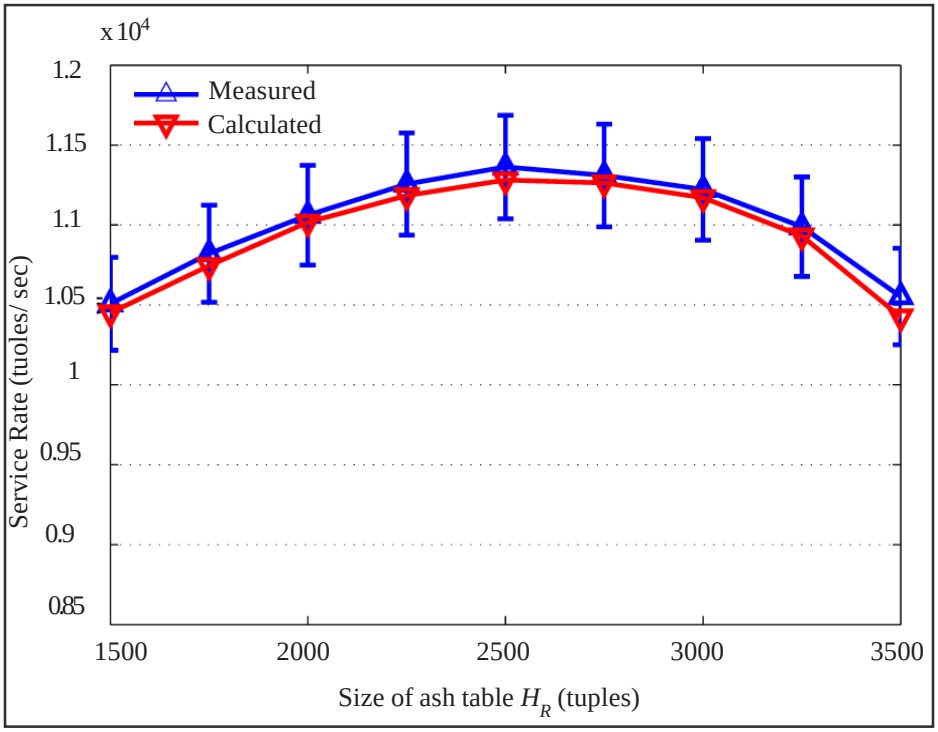


Figure 4. Analysis of w_S and w_N by varying the size of different components



(a) Tuning of disk buffer

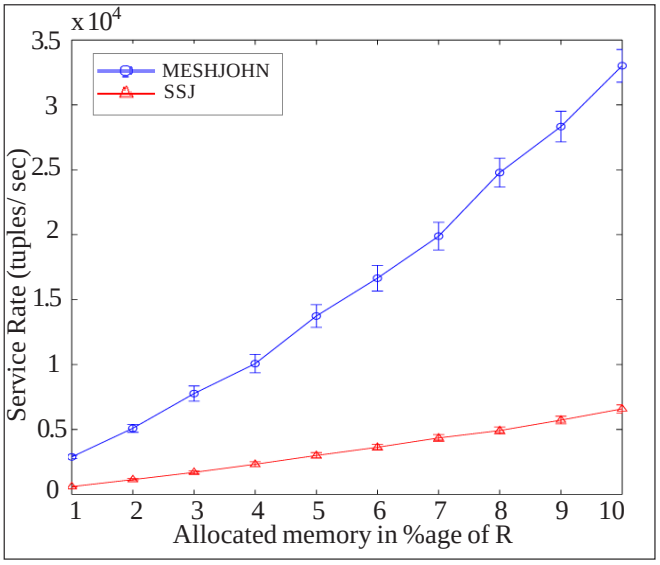


(b) Tuning of hash table H_R

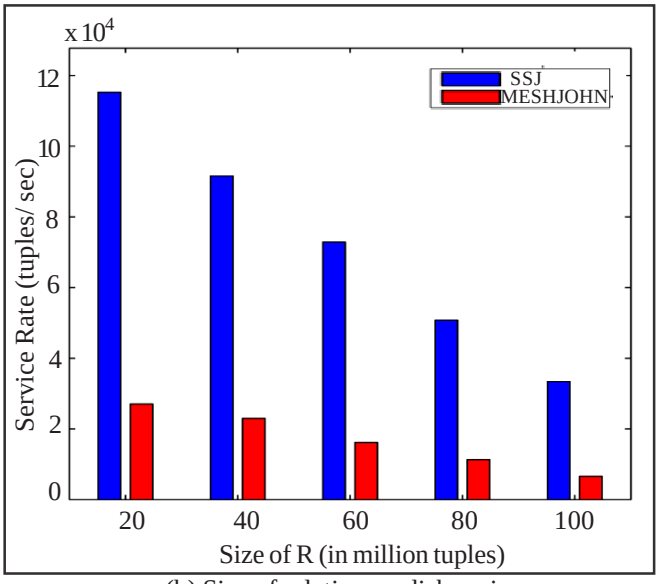
Figure 5. Tuning Comparison: empirical approach vs analytical approach

of the join is measured by calculating the number of tuples processed in a unit second. In each experiment both algorithms first complete their warm-up phase before starting the actual measurements. These kind of algorithms normally need a warm-up phase to tune their components with respect to the available memory resources so that each component can deliver maximum performance. In our experiments, for each measurement we calculate the confidence interval by considering 95% accuracy, but sometimes the variation is very small. We use constant stream arrival rate throughout a run in order to measure the service rate for both algorithms.

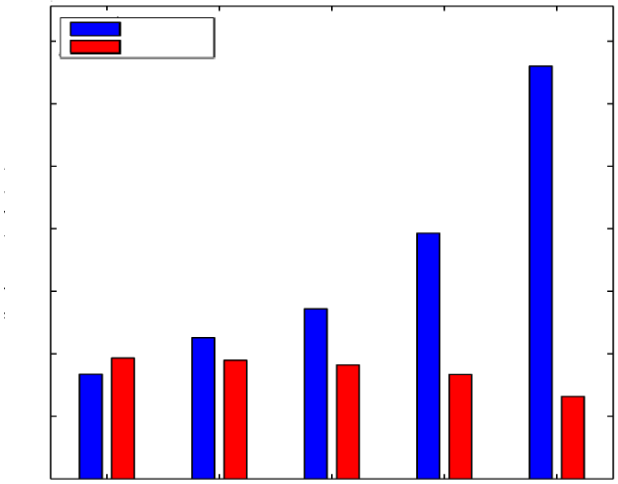
Synthetic data: The stream dataset we used is based on the Zipfian distribution. We test the performance of all the algorithms by varying the skew value from 0 (fully uniform) to 1 (highly skewed). The detailed specifications of our synthetic dataset are shown in Table 2.



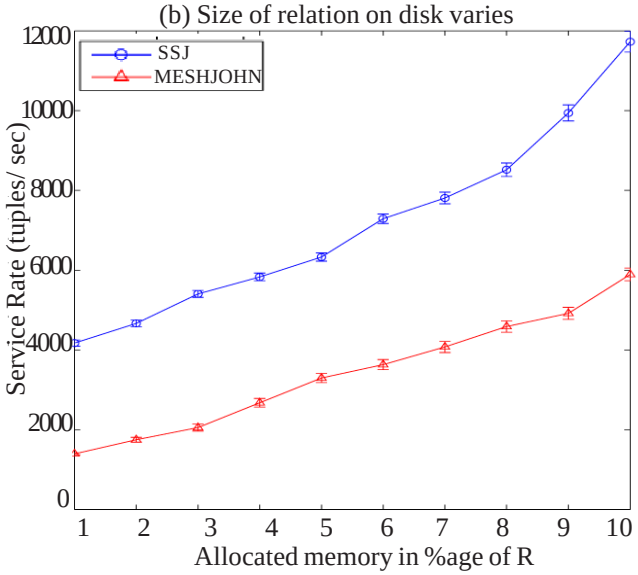
(a) Size of allocated memory varies



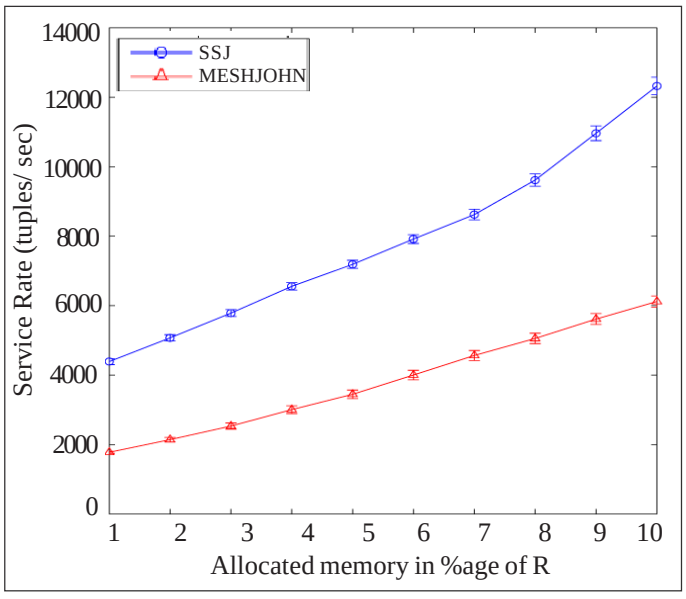
(b) Size of relation on disk varies



(c) Skew in data stream varies

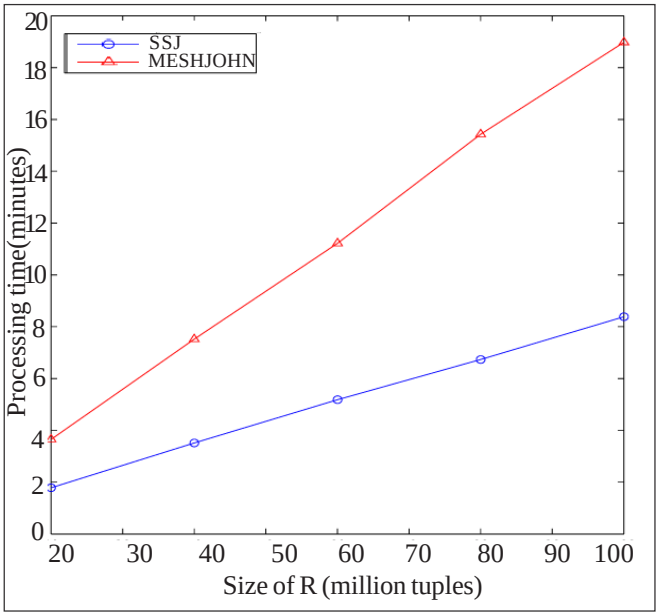


(d) TPC-H dataset

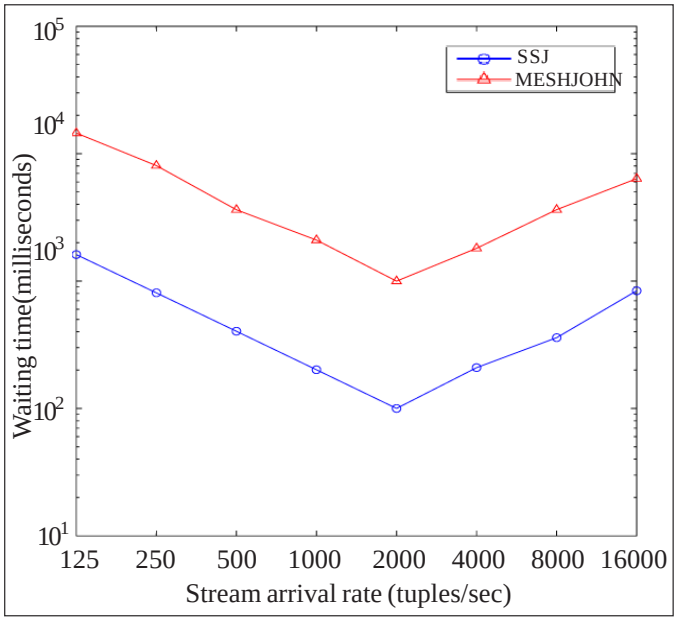


(e) Real-life dataset

Figure 6. Performance analysis by varying external parameters



(a) Processing time



(b) Waiting time

Figure 7. Time analysis

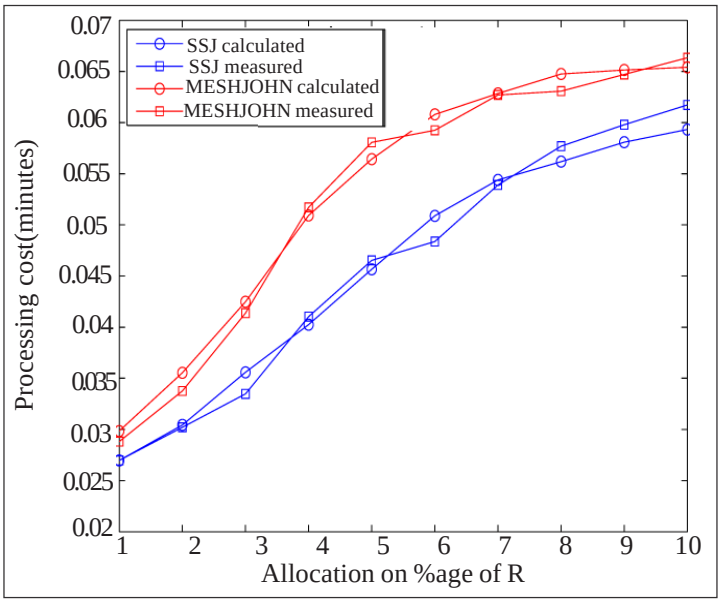


Figure 8. Cost validation

TPC-H: We also analyze the performance of all the algorithms using the TPC-H dataset which is a well-known decision support benchmark. We create the datasets using a scale factor of 100. More precisely, we use table *Customer* as our master data table and table *Order* as our stream data table. In table *Order* there is one foreign key attribute *custkey* which is a primary key in Customer table. So the two tables are joined using attribute custkey. Our *Customer* table contains 20 million tuples while the size of each tuple is 223 bytes. On the other hand *Order* table also contains the same number of tuples with each tuple of 138 bytes. The plausible scenario for such a join is to add customer details corresponding to his order before loading it to the warehouse.

Real-life data: Finally, we also compare the performance of all the algorithms using a real-life dataset[1]. This dataset basically contains cloud information stored in summarized weather reports format. The same dataset was also used with the original MESHJOIN. The master data table contains 20 million tuples, while the streaming data table contains 6 million tuples. The size of each tuple in both the master data table and the streaming data table is 128 bytes. Both the tables are joined using a common attribute, *longitude* (LON), and the domain for the join attribute is the interval [0,36000].

6.2 Performance Evaluation

In this section we present a series of experimental

[1]This dataset is available at: http://cdiac.ornl.gov/ftp/ndp026b/

Parameter	value
Size of disk-based relation R	100 million tuples (≈11.18GB)
Total allocated memory M	1% of R (≈0.11GB) to 10% of R (≈1.12GB)
Size of each disk tuple	120 bytes (similar to MESHJOIN)
Size of each stream tuple	20 bytes (similar to MESHJOIN)
Size of each node in the queue	12 bytes (similar to MESHJOIN)

Table 2. Data specification

comparisons between SSJ and MESHJOIN using synthetic, TPC-H, and real-life data. In our experiments we perform three different analyses. In the first analysis, we compare service rate, produced by each algorithm, with respect to the externally given parameters. In the second analysis, we present time comparisons, both processing and waiting time, for both the algorithms. Finally, in our last analysis we validate our cost models for each algorithm.

External parameters: We identify three parameters, for which we want to understand the behavior of the algorithms. The three parameters are: the total memory available M, the size of the master data table R, and the skew in the stream data. For the sake of brevity, we restrict the discussion for each parameter to a one dimensional variation, i.e. we vary one parameter at a time. Analysis by varying size of memory M: In our first experiment we compare the service rate produced by both the algorithms by varying the memory size M from 1% to 10% of R while the size of R is 100 million tuples (≈11.18GB). The results of our experiment are presented in Figure 6 (a). From the figure it can be noted that SSJ performs up to 7 times faster than MESHJOIN in case of 10% memory setting. While in the case of a limited memory environment (1% of R) SSJ still performs up to 5 times better than MESHJOIN that makes it an adaptive solution for memory constraint applications.

Analysis by varying size of R: In this experiment we compare the service rate of SSJ with MESHJOIN at different sizes of R under fixed memory size, ≈1.12GB. We also fix the skew value equal to 1 for all settings of R. The results of our experiment are shown in Figure 6(b). From the figure it can be seen that SSJ performs up to 3.5 times better than MESHJOIN under all settings of R.

Analysis by varying skew value: In this experiment we compare the service rate of both the algorithms by varying the skew value in the streaming data. To vary the skew, we vary the value of the Zipfian exponent. In our experiments we allow it to range from 0 to 1. At 0 the input stream S is completely uniform while at 1 the stream has a larger skew. We consider the sizes of two other parameters, memory and R, to be fixed. The size of R is 100 million tuples (≈11.18GB) while the available memory is set to 10% of R (≈1.12GB). The results presented in Figure 6(c) show that SSJ again performs significantly better than MESHJOIN even for only moderately skewed data. Also this improvement becomes more pronounced for increasing skew values in the streaming data. At skew value equal to 1, SSJ performs about 7 times better than MESHJOIN. Contrarily, as MESHJOIN does not exploit the data skew, its service rates actually decrease slightly for more skewed data, which is consistent to the original MESHJOIN findings. We do not present data for skew value larger than 1, which would imply short tails. However, we predict that for such short tails the trend continues. SSJ performs slightly worse than MESHJOIN only in a case when the stream data is completely uniform. In this particular case the streamprobing phase does not contribute considerably while on the other hand random access of R influences the seek time.

TPC-H and real-life datasets: We also compare the service rate of both the algorithms using TPC-H and reallife datasets. The details of both datasets have already been described in Section VI-A. In both experiments we measure the service rate produced by both the algorithms at different memory settings. The results of our experiments using TPCH and real-life datasets are shown in Figures 6 (d) and 6 (e) respectively. From the both figures it can be noted that the service rate in case of SSJ is remarkably better than MESHJOIN.

Time analysis: A second kind of performance parameter besides service rate refers to the time an algorithm takes to process a tuple. In this section, we analyze both waiting time and processing time. *Processing time* is an average time that every stream tuple spends in memory from loading to matching without including any delay due to a low arrival rate of the stream. *Waiting time* is the time that every stream tuple spends in the stream buffer before entering into the join module. The waiting times were measured at different stream arrival rates. The experiment, shown in Figure 7 (a), presents the comparisons with respect to the processing time. From the figure it is clear that the processing time in case of SSJ is significantly smaller than MESHJOIN. This difference becomes even more pronounce as we increase the size of R. The plausible reason for this is that in SSJ a big part of stream data is directly processed through the streamprobing phase without joining it with the whole relation R in memory.

In the experiment shown in Figure 7 (b) we compare the waiting time for each of the algorithm. It is obvious from the figure that the waiting time in the case of SSJ is again significantly smaller than MESHJOIN. The reason behind

this is that in SSJ since there is no constraint to match each stream tuple with the whole of R, each disk invocation is not synchronized with the stream input.

Cost analysis: The cost models for both the algorithms have been validated by comparing the calculated cost with the measured cost. Figure 8 presents the comparisons of both costs for each algorithm. The results presented in the figure show that for each algorithm the calculated cost closely resembles the measured cost, which proves the correctness of our cost models.

7. Conclusions

In this paper we discuss a new semi-stream join called SSJ that can be used to join a stream with a disk-based, slow changing master data table. We compare it with MESHJOIN, a seminal algorithm that can be used in the same context. SSJ is designed to make use of skewed, non-uniformly distributed data as found in real-world applications. In particular we consider a Zipfian distribution of foreign keys in the stream data. Contrary to MESHJOIN, SSJ stores these most frequently accessed tuples of R permanently in memory saving a significant disk I/O cost and accelerating the performance of the algorithm. We have provided a cost model of the new algorithm and validated it with experiments. We have provided an extensive experimental study showing an improvement of SSJ over the earlier MESHJOIN algorithm.

References

[1] Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, U., Spatscheck, O. (2002). Gigascope: High performance network monitoring with an SQL interface, *In*: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '02. New York, NY, USA: ACM, p. 623–623. [Online]. Available: http://doi.acm.org/10.1145/564691.564777.

[2] Madden, S., Franklin, M. (2002). Fjording the stream: An architecture for queries over streaming sensor data, *In*: Proceedings of 18th International Conference on Data Engineering. IEEE, p. 555–566.

[3] Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., Strauss, M. (2001). Surfing wavelets on streams: One-pass summaries for approximate aggregate queries, *In*: Proceedings of the 27th International Conference on Very Large Data Bases, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., p. 79–88. [Online]. Available: http://portal.acm.org/citation.cfm?id=645927.672174.

[4] Arasu, A., Babu, S., Widom, J. (2002). An abstract semantics and concrete language for continuous queries over streams and relations, Stanford InfoLab, Technical Report 2002-57. [Online]. Available: http://ilpubs.stanford.edu:8090/563/

[5] Wu, E., Diao, Y., Rizvi, S. (2006). High-performance complex event processing over streams, *In*: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '06. New York, NY, USA: ACM, p. 407–418. [Online]. Available: http: //doi.acm.org/10.1145/1142473.1142520

[6] Naeem, M. A., Dobbie, G., Weber, G. (2008). An event-based near real-time data integration architecture, *In*: EDOCW '08: Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops. Washington, DC, USA: IEEE Computer Society, p. 401–404.

[7] Golab, L., Johnson, T., Seidel, J. S., Shkapenyuk, V. (2009). Stream warehousing with datadepot, in SIGMOD '09: *In*: Proceedings of the 35th SIGMOD International Conference on Management of Data. New York, NY, USA: ACM, p. 847–854.

[8] Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N. (2008). Meshing streaming updates with persistent data in an active data warehouse, *IEEE Trans. on Knowl. and Data Eng.*, 20 (7) 976–991.

[9] Supporting streaming updates in an active data warehouse, *In*: ICDE 2007: Proceedings of the 23rd International Conference on Data Engineering, Istanbul, Turkey, p. 476–485.

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

[11] Wilschut, A. N., Apers, P. M. G. (1990). Pipelining in query execution, *In*: Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications (PARBASE 1990), Miami Beach, FL, USA. Los Alamitos: IEEE Computer Society Press, March 1990, p. 562–562.

[12] Lawrence, R. Early Hash Join: A configurable algorithm for the efficient and early production of join results, *In*: VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases. VLDB Endowment, p. 841–852.

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