Comparing Global Optimization and Default Settings of Stream-based Joins

M. Asif Naeem, Gillian Dobbie, and Gerald Weber

Department of Computer Science, The University of Auckland,
Private Bag 92019, 38 Princes Street, Auckland, New Zealand
mnae006@aucklanduni.ac.nz,{gill, gerald}@cs.auckland.ac.nz

Abstract. One problem encountered in real-time data integration is the join of a continuous incoming data stream with a disk-based relation. In this paper we investigate a stream-based join algorithm, called mesh join (MESHJOIN), and focus on a critical component in the algorithm, called the disk-buffer. In MESHJOIN the size of disk-buffer varies with a change in total memory budget and tuning is required to get the maximum service rate within limited available memory. Until now there was little data on the position of the optimum value depending on the memory size, and no performance comparison has been carried out between the optimum and reasonable default sizes for the disk-buffer. To avoid tuning, we propose a reasonable default value for the disk-buffer size with a small and acceptable performance loss. The experimental results validate our arguments.

Key words: ETL for real-time data warehouse, ETL optimization, Tuning and management of the real-time data warehouse, Performance and scalability, Stream-based join

1 Introduction

Data warehouses are used to aggregate and analyze data in order to provide reliable information for business-related decisions. Traditional data warehouses are static data repositories based on batch-driven Extract-Transform-Load (ETL) tools. Data loading from operational systems to data warehouses is usually performed on a nightly basis or even in some cases on a weekly basis, therefore typical data warehouses do not have the most current data.

To make data integration near to real-time a new concept of data warehousing, called real-time or active data warehousing, has been introduced [1][2][3][4][5][6]. In real-time data warehousing the updates occurring on operational data sources are immediately reflected in the data warehouse.

In data warehousing, the data needs to be transformed into the destination format before loading it into the data warehouse. One important type of transformation is the replacement of the key in the data source with a data warehouse key, which is also called a surrogate key. A surrogate key is typically an integer-based format which is used to access the data from the data warehouse uniquely.

To perform this key transformation in practice a join is required between source updates and a master data table.
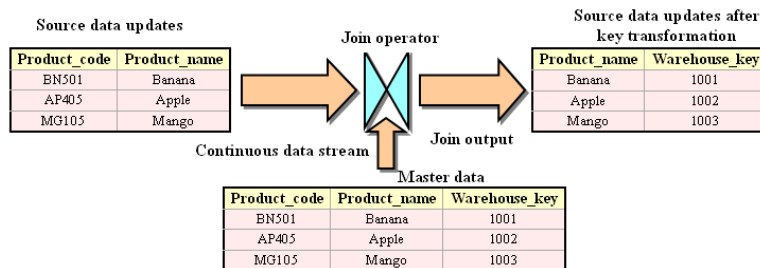

Fig. 1. An example of key transformation

Consider the example shown in Figure 1 where the updates occurring at the source level are propagated to the transformation layer. In the transformation phase, the source keys are validated against the master data using a join and the corresponding records are enriched producing output tuples.

In proprietary data warehousing, this join is usually performed using block algorithms [15][7] where the incoming tuples are buffered and then joined in order to reduce the execution time. In the case of real-time data warehousing where a continuous stream of updates is propagated and needs to be processed, these algorithms do not work efficiently because of the blocking factor. The major challenge is to deal with the different rates of join inputs. The incoming stream is fast while the access rate of disk-based tuples is relatively slow.

The Index-Nested-Loop (INL) algorithm is an option for implementing this join. In the case of INL, a stream S is scanned tuple by tuple and the look-up relation R is accessed using a cluster-based index on the join attribute. Although this join algorithm can deal with a continuous stream and can generate the output in an online fashion, it requires extra time to maintain an index on the join attribute and also it handles one tuple at a time reducing the throughput.

The MESHJOIN algorithm was introduced by Polyzotis et al. [5][6] to perform joins between a continuous stream and a disk-based relation using limited memory. The key idea of the algorithm is that, for each iteration it retrieves a number of pages from disk and a set of tuples from the stream and then loads them into relevant buffers. The disk buffered pages are then joined with all tuples stored in a hash table and the final output is generated. At the start of next iteration the expired tuples are discarded from the hash table and the new inputs are scanned into both buffers. The advantage of this algorithm is to amortize the fast arrival rate of the incoming stream by executing the join of disk pages with a large number of stream tuples.

Although the original paper gave a very clear explanation of how the algorithm worked, it contained only a very brief evaluation of one of its critical components, the disk-buffer, which stores the disk relation R. For every new

memory budget, MESHJOIN tunes this disk-buffer size in order to find its optimum value. This is further explained in section 3.

In this paper we evaluate the algorithm and propose an alternative to the tuning approach for the MESHJOIN algorithm. We analyze the performance of the algorithm for different sizes of disk-buffer, and compare the performance for the optimal disk-buffer size with the performance for a default size that remains constant for all memory budgets. We find a difference of less than two percent. In the straightforward implementation of MESHJOIN, the tuning component has full control over the buffer size. Since the tuning component has a sizeable code base, it can have errors. A typical estimate assumes 20 errors per 1000 lines of code [22]. These errors can produce widely deviating buffer sizes, or worse fatal errors. Widely deviating buffer sizes are a higher risk than the default size. Therefore our findings suggest that in critical applications the tuning component could be omitted and the default size should be chosen.

The rest of the paper is structured as follows. Section 2 focuses on the working, architecture and algorithm for MESHJOIN. Our observations about MESHJOIN are discussed in section 3. Tuning and performance comparisons using default and optimum values for the disk-buffer sizes, are presented in section 4. Section 5 explains the strategy for choosing the default value for the size of disk-buffer. Section 6 describes the related work and finally section 7 concludes the paper.

2   MESHJOIN

To support real-time data warehousing, the stream-based algorithm MESHJOIN is designed for joining a fast stream S with a large disk-based relation R under a limited memory budget. The algorithm can be tuned to maximize the output for a specific allocated memory size or to minimize the memory limit for a specific output. The authors made the following assumptions about the join input parameters for the MESHJOIN algorithm:

1. The disk-based relation R remains unchanged during the transformation.
2. There are no special physical characteristics (e.g. index or cluster property) of R.
3. The algorithm receives a continuous stream S from the data source without any bottleneck.

2.1   Basic operation

The operation of MESHJOIN is illustrated with the help of an example [5][6]. Assume that R contains two pages, p_1 and p_2, and there is sufficient memory to hold the window of the two most recent tuples of the stream. The operation of the algorithm at different time intervals is depicted in Figure 2.

1. At time $t=0$, the algorithm scans the first stream tuple s_1 and the first page p_1 from relation R and joins them in memory.
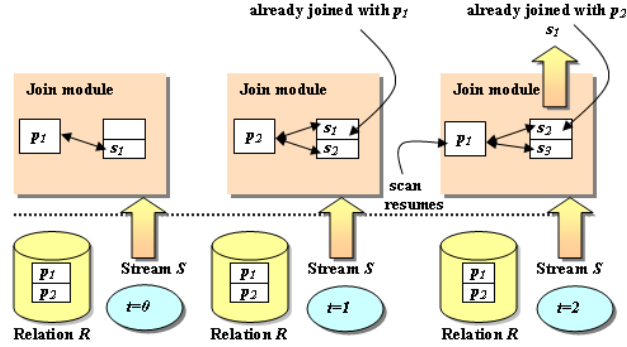
Fig. 2. MESHJOIN Operation[5][6]

2. At time $t=1$, the algorithm brings a second stream tuple s_2 into memory along with the second page p_2. Currently the page p_2 is joined with two stream tuples. Since the stream tuple s_1 has been joined with all of relation R it can be removed from memory.
3. At time $t=2$, the algorithm again retrieves both inputs p_1 and s_3 into memory from the sources R and S respectively. At this time page p_2 is replaced by p_1 and s_1 with the next stream tuple s_3 and thus page p_1 is joined with s_2 and s_3. As stream tuple s_2 has joined with both pages p_1 and p_2, it is discarded from memory.

2.2   Architecture

The architecture of the MESHJOIN algorithm is shown in Figure 3. In the figure, there are two input sources, one is a continuous data stream S and the other is a disk-based relation R. MESHJOIN continuously scans the data from these input sources and joins them together in order to generate the result. The disk relation R is scanned sequentially but in a cyclic manner i.e. after scanning the last page it again starts from the first page. It is assumed that k iterations are required to bring the whole relation R into memory. In each iteration a set of tuples w is scanned from stream S and stored in the hash table H along with their pointer addresses in a queue Q. The size of Q in terms of number of partitions is normally equal to the number of iterations k. The reason is once the stream tuples enter into the execution window they are probed by all tuples of relation R before they expire. The key function of Q is to keep a record of stream tuples in order to identify the expired tuples for each iteration. In each iteration MESHJOIN scans b pages from disk and loads them into a buffer, therefore the total number of pages in R is $N_R = k.b$.

2.3   Algorithm

The pseudo-code for the original algorithm is shown in Figure 4. For each iteration, the algorithm takes two parameters, w tuples and b pages, from the
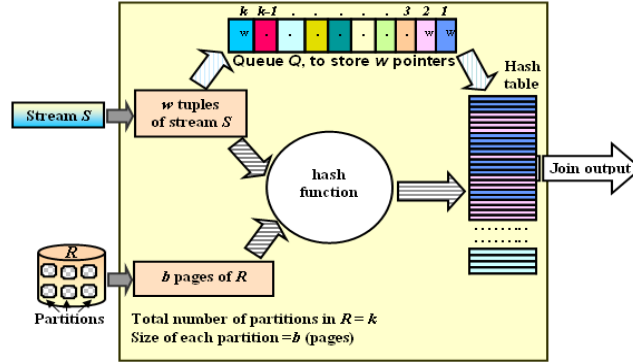
Fig. 3. Architecture of MESHJOIN[5][6]

input sources S and R respectively and feeds them into relevant buffers. Before starting the join execution the algorithm monitors the status of Q. If it is already full, the algorithm dequeues the pointer addresses of the oldest tuples and discards the corresponding tuples from the hash table. In the next step the algorithm stores w tuples in the hash table with their corresponding addresses into Q. Finally, it generates the required output after performing the join of b pages with all tuples in the hash table.

MESHJOIN algorithm

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

Fig. 4. MESHJOIN algorithm[5][6]

3   Problem definition

In this paper we focus on an optimization problem for a critical component of MESHJOIN. As shown in Figure 3 , the disk-buffer component of MESHJOIN is used to load a group of disk pages into memory and its size varies with a change in the total allocated memory for join execution. Therefore, in order to achieve the maximum service rate within a fixed memory budget, MESHJOIN first tunes that disk-buffer component. The parameters that MESHJOIN uses in tuning are based on a cost model.

To explore the analytical steps behind this tuning process we consider the cost equations [5][6], both in terms of memory and processing, used by MESHJOIN. To calculate the memory cost, MESHJOIN uses the following equation while the symbols used in the cost equations are explained in Table 1.

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

where M is the total memory reserved by all join components, which can be less than or equal to the maximum memory budget, $b \cdot v_P$ is the piece of memory allocated for the disk-buffer, $w \cdot v_S$ is the memory reserved for the stream-buffer, $w \frac{N_R}{b} \text{sizeof}(ptr)$ represents the memory reserved by the queue and finally, $w \cdot f \frac{N_R}{b} v_S$ is the memory allocated for the hash table.

The MESHJOIN processes w tuples in each iteration of the algorithm and the processing cost for one iteration is denoted by c_{loop} that can be calculated using the following equation.

$$c_{loop} = c_{I/O}(b) + w \cdot c_E + w \cdot c_S + w \cdot c_A + b \frac{v_P}{v_R} c_H + \sigma b \frac{v_P}{v_R} c_O \qquad (2)$$

where $c_{I/O}(b)$ is the cost to read b pages from the disk, $w \cdot c_E$ is the cost to expire w tuples from the queue and hash table, $w \cdot c_S$ is the cost to read w tuples from stream S into the stream-buffer, $w \cdot c_A$ represents the cost to append w tuples into the queue and the hash table, $b \frac{v_P}{v_R} c_H$ denotes the probing cost of all tuples in b pages into the hash table, and finally, $\sigma b \frac{v_P}{v_R} c_O$ represents the cost to generate output for b pages.

Also equation (1) can be written in the following form as:

$$w = \frac{M - b \cdot v_P}{v_S + \frac{N_R}{b} \text{sizeof}(ptr) + \frac{N_R}{b} v_S \cdot f} \qquad (3)$$

Similar to equation (2), c_{loop} is the processing cost for w tuples therefore, the service rate μ can be calculated using the following equation.

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

By substituting the value of w in equation (4),

$$\mu = \frac{M - b \cdot v_P}{c_{loop}(v_S + \frac{N_R}{b} \text{sizeof}(ptr) + \frac{N_R}{b} v_S \cdot f)} \qquad (5)$$

Table 1. Notations used in cost estimation of MESHJOIN

Parameter name	Symbol
Size of each tuple of S (bytes)	v_S
Number of pages in R	N_R
Size of each tuple in R (bytes)	v_R
Size of each page in R (bytes)	v_P
Number of pages of R in memory for each iteration	b
Total number of iterations required to bring the whole R into memory	k
Number of stream tuples read into join window for each loop iteration	w
Cost of reading b disk pages into the disk-buffer	$c_{I/O}(b)$
Cost of removing one tuple from H and Q	c_E
Cost of reading one stream tuple into the stream-buffer	c_S
Cost of appending one tuple into H and Q	c_A
Cost of probing one tuple into the hash table	c_H
Cost to generate the output for one tuple	c_O
Total cost for one loop iteration of MESHJOIN (seconds)	c_{loop}
Total memory used by MESHJOIN (bytes)	M
service rate (tuples/second)	μ

If we are interested in the maximum service rate depending on b, then we can find the maximum of equation (5) as a function of b using numerical methods. Numerical methods are necessary, since equation (5) depends on $c_{I/O}$, which is a measured function of b and we have no analytical formula for it. MESHJOIN uses a tuning step, where for each memory budget M, the optimal disk buffer size b is determined by solving this numerical problem. The size of the disk-buffer is not fixed and a tuning effort is made for every new memory budget. The issue is whether this tuning effort is really necessary.

4   Tuning and performance comparisons

4.1   Proposed investigation

We decided that in order to assess the necessity of the tuning process for the component disk-buffer in MESHJOIN, we need empirical results about how the cost function behaves in a real world scenario, and how much better the performance for the optimal setting is, as compared to reasonable default settings. Since the original code was not available on request, we investigate this problem by implementing the MESHJOIN algorithm ourselves, incorporating the same assumptions around the input stream and disk-based relation R as described in section 2. Our implementation and settings are available for download.

As a preview of our findings in this paper and to indicate where we are heading, we show in Figure 5 a sample performance measurement of MESHJOIN for different sizes of the disk-buffer within a fixed memory budget. Note that in order to magnify the effect under investigation the y-axis does not start with zero . We observe that the curve has a pronounced knee [21]. According to
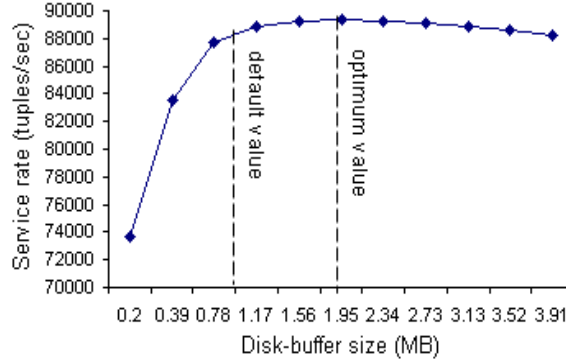
Fig. 5. Effect of disk-buffer on MESHJOIN performance using fixed memory budget (80MB)

the figure the service rate grows drastically up to the knee in the curve. We observe a saturation behavior, where incrementing the disk-buffer size improves the performance only a little. This is important, because it allows us to choose a default value near the knee of the curve. In the end, we will come up with a reasonable default value for the disk-buffer size that holds for a series of memory budgets. Before proceeding towards the experimental results we first describe the experimental setup.

4.2   Experimental setup

We implemented a prototype of the MESHJOIN algorithm using the following specifications.

Hardware specifications: We conducted our experiment using Pentium-IV 2x2.13GHz machine with 3G main and 160G disk memory under Windows-XP. We implemented the experiment in Java using the Eclipse IDE Version: 3.3.1.1. We also used built-in plug-ins, provided by Apache, and built-in functions like *nanoTime()*, provided by the Java API, to measure the memory and processing time. In addition to that, Java hash table does not support the storage of multiple tuples against one key value. To resolve this issue we used multi-hash-map, provided by Apache, in our experiments.

Data specifications: We analyzed the performance of MESHJOIN using synthetic data. The look-up data (relation R) is stored on disk while the stream data is generated at run time using our own random-number generating procedure. Both the look-up data file and random number generating procedure are also available along with our open source MESHJOIN implementation. We tested our experiment with varying sizes of disk-buffer to find its optimum default value. On the other hand the size of the stream-buffer is flexible and fluctuates with the size of disk-buffer. Similarly the size of the Q (in terms of partitions) also varies with the total number of iterations required to bring the whole R

into disk-buffer. The detailed specification of the data that we used for analysis is shown in Table 2.

System of measurement: The performance of the join is measured by calculating the number of tuples processed in a unit second, which is the service rate and is denoted by μ. We start our measurement after some iterations of the loop. For increased accuracy we take three readings for each specification and take their average. Moreover, it is assumed that during the execution of the algorithm no other applications run in parallel.

Table 2. Experimental data characteristics

Parameter	value
Disk-based data	
Size of disk-based relation R	3.5 millions tuples
Size of each tuple	120 bytes
Default size for the disk-buffer	0.93MB
Stream data	
Size of each tuple	20 bytes
Size of each pointer in Q	4 bytes
Fudge factor for hash table	4.8

4.3 Tuning of disk-buffer for different memory budgets

We first analyze the optimum values of the disk-buffer size for a series of memory budgets and the join performance at these optimum values. No such values for different memory settings have been published before. In order to obtain the optimum value for the disk-buffer size we tuned MESHJOIN for a series of memory budgets.

Figure 6 depicts the optimum values for the disk-buffer size in the case of different memory budgets. The figure shows that the size of disk-buffer increases with an increase in the total memory budget. As the total memory M depends on w and b and we also stated that w also depends on b, therefore the optimum size of disk-buffer b will increase with an increase in the total memory budget.

4.4 Performance analysis using default and optimum values for the disk-buffer size

In this experiment we test the MESHJOIN algorithm for a series of memory budgets in order to observe the real difference in performance for a reasonable default value and optimum values of the disk-buffer size. Figure 7 shows performance measurements for different memory budgets along with the default and optimum values for disk-buffer size. Note that, the scales in Figure 7 differ from Figure 10(a). The scale of the y-axis is larger in Figure 7, and only the lowest
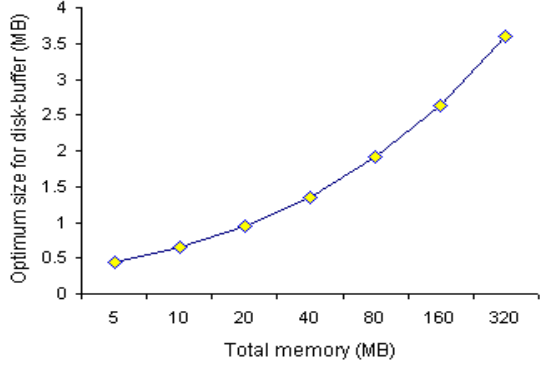
Fig. 6. Optimum values for the disk-buffer size with respect to the different memory budgets
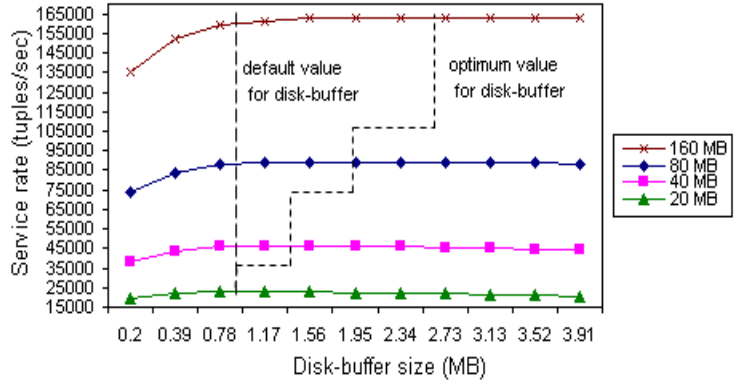

Fig. 7. Performance comparisons using default and optimum values for the disk-buffer size in case of different memory budgets

most curve is shown in Figure 10(a). Also in Figure 5, only the 80MB curve is shown.

The optimum value for 20MB is 0.93MB. The setting 20MB is the memory budget from the original MESHJOIN paper and for today's computing landscape a very small value for a server component even when considering limited memory budgets. For the purposes of this discussion we deemed it most helpful to use the optimum value for this setting as the default value, because if we obtain a reasonable performance for all other memory budgets, there is a strong indication that tuning dependent on the overall memory budget is not necessary.

We observe for all memory budgets a clear saturation behavior. In the case of 40MB as total memory budget, the value for the optimum disk-buffer size is 1.35MB and the improvement in performance as compare to the default size of disk-buffer is only 0.4%. By considering the 80MB total memory budget, the

value for the optimum disk-buffer size is 1.91MB with 1.17% performance improvement. Finally, in the case of 160MB as total memory budget, the optimum value for the disk-buffer size is 2.63MB and it again improves the performance a little, 1.78%.
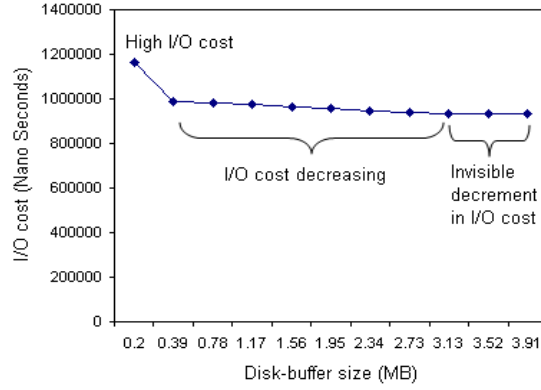

Fig. 8. Disk I/O cost for different sizes of disk-buffer

To prove this experimentally we measured the I/O cost per page amortized over all pages read into the disk-buffer in one iteration. The per page I/O cost for different sizes of disk-buffer is depicted in Figure 8. The figure shows that in the beginning the I/O cost is high due to the small size of the disk-buffer. After that as the size of disk-buffer increases the amortized I/O cost per page decreases, but after a while further increments in the size of the disk-buffer does not reduce the I/O cost considerably.
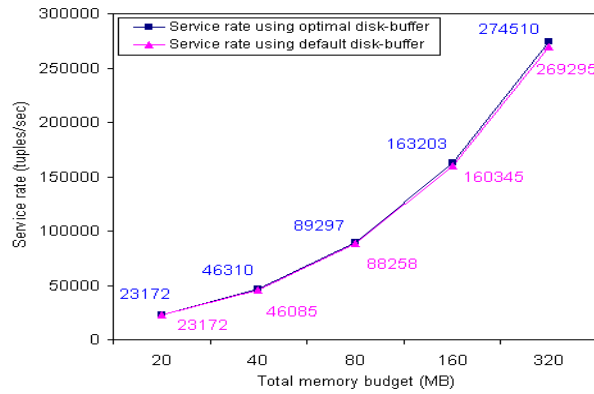

Fig. 9. Performance comparison directly at default and optimum values of disk-buffer size using different memory budgets

To visualize the performance difference more clearly we also measure the MESHJOIN performance directly on the default value and the optimum values of disk-buffer size for a series of memory budgets. Figure 9 depicts the experimental results in both cases. From the figure it is clear for small memory budgets the performance of the algorithm is approximately equal, and even for a large memory limit (320MB) there is no remarkable improvement in performance.

4.5   Cost validation

In this section we validate our implementation of MESHJOIN by comparing the predicted cost with the measured cost. In the case of the predicted cost, we first calculated the cost for one loop iteration using equation (2) and then calculated the service rate by applying the formula in equation (4). To validate the cost model we performed two different kinds of experiments. The results of both experiments are shown in Figure 10.
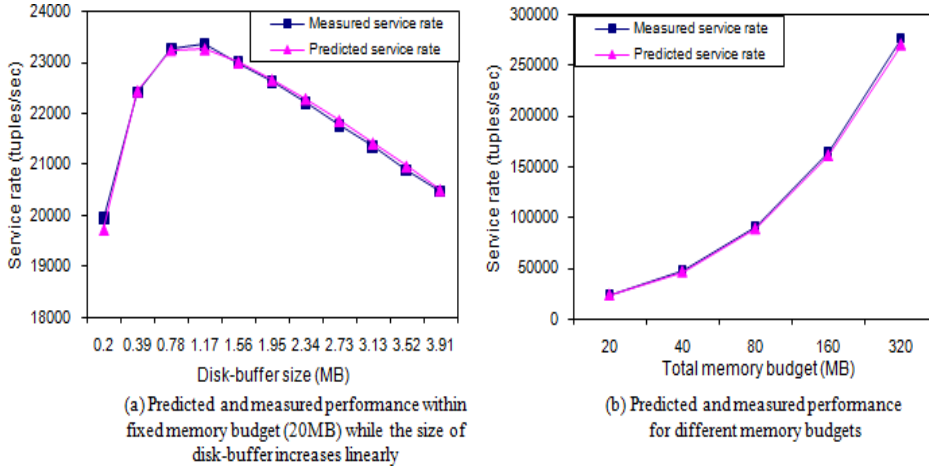

(a) Predicted and measured performance within fixed memory budget (20MB) while the size of disk-buffer increases linearly

(b) Predicted and measured performance for different memory budgets

Fig. 10. Cost validation of MESHJOIN

In our first experiment, shown in Figure 10(a), the size of disk-buffer increases linearly while the total memory budget is fixed. In the figure both the measured and the predicted results indicate that the performance of MESHJOIN remains consistently high for small values of disk-buffer size and drops rapidly as disk-buffer size is increased. In our second experiment, shown in Figure 10(b), we validate the cost model using optimum disk-buffer for different memory budgets. However, in both cases the measured cost closely resembles the predicted cost, validating the correctness of our MESHJOIN implementation.

5   Approach for choosing the default value

Although the function for the performance of MESHJOIN depending on the buffer size has a pronounced knee (see Figure 5), it is still a smooth curve, therefore the question arises, which exact value should be chosen as a default value. Through practice, we have observed that a value for the buffer size which is optimal for a specific setting is at the same time still a good value for a wide range of settings, and therefore suitable as a default value.

In order to support this we have chosen in our experiments a default size of 0.93MB, which is the optimal buffer size for a very small memory budget. Our experiments have shown that this setting is also sufficient for other memory budgets allocated for MESHJOIN. In particular we have shown that the results for this default value were less than 3 percent below the optimum for all tested memory settings. We restricted our tests to memory sizes up to 320MB. This restriction is motivated by the fact that MESHJOIN, according to the authors of the original publication, is designed for a limited memory budget. In fact the original publications only consider memory budgets up to 40MB, so our investigation up to 320MB has sufficient security margin. In summary our experiments have shown that while the optimal disk buffer size varies over a certain range, the performance achieved with them varies only in the order of a few percent. Therefore, in settings where simplicity of the system has precedence over very small performance gains, the default buffer size strategy seems worthwhile.

This default value is still dependent on the underlying hardware, therefore we focus primarily on the transferability of default values for settings on the same hardware. Nevertheless it is fair to assume that even across different but similar hardware configurations there will be some transferability.

6   Related work

Most of the research related to data warehousing deals with managing proprietary warehouses [8][9][12]. To make data integration near to real-time, different approaches [1][2][3][4] have been introduced that primarily use block algorithms [11][10] to perform the join between source updates and the look-up table. These block algorithms store the incoming data streams in disk-buffers and process them in batches. Therefore, such algorithms normally can work efficiently for off-line data loading windows.

In the field of real-time data warehousing the processing of continuous data streams has become an emerging area for research. Researchers have explored the area from different perspectives and inspected issues related to join execution requirements, data models, challenges in query processing and different algorithmic problems [13][14].

The sort-merge join [15] is a well known algorithm that joins two different data sets. The major drawback with this algorithm is that it cannot start its execution until all the data is sorted, causing unnecessary delays to generate the join output.

To remove this delay different progressive joins are proposed [16][17][18][19]. The basic aim of these algorithms is to generate the output as fast as the tuples arrive. The key idea used in these approaches is to access the input stream in a continuous manner and in the case when memory is not sufficient, the excess tuples are flushed to the disk to be processed later when resources are free. The key challenge with these approaches is the need to process each tuple very efficiently while there is a large volume of incoming data. In addition to that the stream amortizing cost should be smaller than the time difference of two contiguous incoming tuples. Under certain conditions, the number of unprocessed tuples will grow regularly and exceed the memory limit.

The novel flushing algorithm [20] was proposed to enhance the performance of a progressive join. Again this algorithm does not fulfil our requirements because the stream tuples are stored on disk rather than in a memory buffer which can be refreshed in an online fashion. The recent algorithm [5][6] that we focus on in this paper, fulfils the requirement of join execution with a continuous data stream.

7   Conclusions and future work

In real-time data warehousing the stream of update tuples needs to be transformed in an online fashion before loading the result into the data warehouse. To perform this transformation a join operator is required in order to probe the incoming stream tuples with master data. In this research we explore a stream-based join, MESHJOIN. MESHJOIN reserves a variable size of memory for a disk-buffer to store the relation R and the procedure to measure the size of the disk-buffer was not previously evaluated. In addition for every memory budget the algorithm tunes the disk-buffer in order to find its optimum value. In our research we defined a complete set of parameter settings for the setup. The example default settings for the setup used here are derived from experimental results. We have shown that the default settings are <2% worse than the optimum, which should be taken into account when considering the importance of the optimization process. Given that the tuning component is a sizeable fraction of the code, and every code can have bugs, this is an important indication that in mission critical systems one should consider only using the default size. We have provided an open source implementation of the MESHJOIN algorithm.

In order to deal with the intermittent nature of the input stream updates, in the future we will extend the implementation of the MESHJOIN algorithm with indexes on the disk-based relation that will further enhance the efficiency of real-time data warehousing.

Source URL: The source for our implementation can be downloaded from. *https://www.cs.auckland.ac.nz/research/groups/serg/mj/BIRTE/*

References

1. Bruckner, R., M., List, B., Schiefer, J.: Striving towards Near Real-Time Data Integration for Data Warehouses. In: DaWaK 2000: Proceedings of the 4th International

Conference on Data Warehousing and Knowledge Discovery, pp. 317–326. Springer-Verlag, London, UK(2002)

2. Nguyen, A., Tjoa, A.: Zero-Latency data warehousing for hetrogeneous data sources and continuous data streams. In: iiWAS'2003 - The Fifth International Conference on Information Integrationand Web-based Applications Services, Austrian Computer Society(OCG)(2003). pp. 55–64

3. Francisco, A.: Real-time Data Warehousing with Temporal Requirements. In: CAiSE Workshops(2003)

4. Karakasidis, A., Vassiliadis, P., Pitoura, E.: ETL queues for active data warehousing. In: IQIS '05: Proceedings of the 2nd international workshop on Information quality in information systems, pp. 28–39. ACM, New York, NY, USA(2005)

5. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N.E.: Supporting Streaming Updates in an Active Data Warehouse. In: ICDE 2007. IEEE 23rd International Conference on Data Engineering, pp. 476-485. Los Alamitos, CA, USA(2007)

6. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N.: Meshing Streaming Updates with Persistent Data in an Active Data Warehouse. In: IEEE Trans. on Knowl. and Data Eng., vol. 20, no. 7, pp. 976–991, Piscataway, NJ, USA(2008)

7. Shapiro, L. D.: Join processing in database systems with large main memories. In: ACM Trans. Database Syst., vol. 11, no. 3, pp. 239–264, New York, NY, USA(1986)

8. Galhardas, H., Florescu, D., Shasha, D., Simon, E.: AJAX: an extensible data cleaning tool. In: SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pp. 590, New York, NY, USA(2000)

9. Labio, W., Yang, J., Cui, Y., Garcia-Molina, H., Widom, J.: Performance Issues in Incremental Warehouse Maintenance. In: VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases, pp. 461–472, San Francisco, CA, USA(2000)

10. Labio, W. J., Wiener, J. L., Garcia-Molina, H., Gorelik, V.: Efficient resumption of interrupted warehouse loads. In: SIGMOD Rec., vol. 29, no. 2, pp. 46–57, New York, NY, USA(2000)

11. Labio, W., Garcia-Molina, H.: Efficient Snapshot Differential Algorithms for Data Warehousing. In: VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases, pp. 63–74, San Francisco, CA, USA(1996)

12. Raman, V., Hellerstein, J. M.: Potter's Wheel: An Interactive Data Cleaning System, In: VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases, pp. 381–390, San Francisco, CA, USA(2001)

13. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 1–16, New York, NY, USA(2002)

14. Golab, L., Özsu, M. T.: Issues in data stream management. In: SIGMOD Rec., vol. 32, no. 2, pp. 5–14, New York, NY, USA(2003)

15. Blasgen, M. W., Eswaran, K. P.: Storage and access in relational data bases. In: IBM System, vol. 16, no. 4, pp. 363, (1977)

16. Mohamed, F. M., Ming, L., Walid, G. A.: Hash-merge Join: A Non-blocking Join algorithm for Producing Fast and Early Join Results. In: ICDE '04: Proceedings of the 20th International Conference on Data Engineering, pp. 251–263, Washington, DC, USA(2004)

17. Tolga, U., Michael, J. F.: Xjoin: A reactively-scheduled pipelined join operator. In: IEEE Data Engineering Bulletin, vol. 23, no.2, pp. 27–33, (2000)

18. Viglas, S. D., Naughton, J. F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: VLDB '2003: Proceedings of the 29th International Conference on Very large Data Bases, pp. 285–296, Berlin, Germany(2003)
19. Dittrich, J., Seeger, B., Taylor, D. S., Widmayer, P.: Progressive merge join: a generic and non-blocking sort-based join algorithm. In: VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases, pp. 299–310, Hong Kong, China(2002)
20. Tao, Y., Yiu, M. L., Papadias, D., Hadjieleftheriou, M., Mamoulis, N.: RPJ: producing fast join results on streams through rate-based optimization. In: SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 371–382, New York, NY, USA(2005)
21. Fogiel, M.: Basic Electricity. Research and Education Assoc. (2002), pp. 355. http://www.flipkart.com/handbook-basic-electricity-research-education/087891420x-c9w3f1iclf#previewbook
22. Gaffney, J. E.: Estimating the number of faults in code. In: IEEE Transactions on Software Engineering. vol. SE-10, no. 4, pp. 459-464. 1984