A Case for Merge Joins in Mediator Systems

Ramon Lawrence   Kirk Hackert

IDEA Lab, Department of Computer Science, University of Iowa
Iowa City, IA, USA
{ramon-lawrence, kirk-hackert}@uiowa.edu

Abstract

There has been considerable work on determining efficient joins for mediator systems. Mediator systems have to dynamically integrate the results of multiple sources in a timely and efficient fashion. Thus, a focus has been on the ability to produce initial results early and to handle network latency and blocking. The issue of limited client capabilities and its impact on join algorithm selection has not been examined. A mediator system may be a large database system in itself or as small as a PDA, cellphone, or embedded Java applet on a web page. It is these smaller, dynamic mediator systems that hold promise of rapid integration of web resources. In this work, we examine several of the existing join methods under the context of supporting thin client mediators which have limited memory, CPU, or disk space. Somewhat surprisingly, the basic sort-merge join applies exceptionally well to this environment with its parallel exploitation of server resources and minimal client resource constraints.

1 Introduction

There are a wide variety of mediator systems for integrating heterogeneous resources. Some mediators are hosted on powerful database servers, while others can be hosted on small personal devices such as PCs, cellphones, and PDAs. Dynamic integration of web sources on thin clients holds the promise of making integration more portable and accessible to users. Most join techniques implicitly assume that the client is capable of efficiently performing the join of data extracted from the sources. This is not always the case with thin clients who have limited CPU, memory, and disk capacities. Even if the mediator is hosted on a powerful machine that serves as an access point for many clients, it may be advantageous to push some of the join processing to the servers instead of performing it at the mediator. This allows the client to take advantage of server computational power to simplify its computation and produce results quicker. The goal is to evaluate the performance of sort-merge join in this environment with respect to other join methods, especially those who produce tuples early (non-blocking join algorithms). The contribution of this work is the first performance comparison of join methods for mediator systems with respect to limited client resources.

This paper is organized as follows. Section 2 provides background on several join methods and a discussion of their CPU, memory, and I/O demands at the mediator and the sources. Section 3 details our experiments and results, and the paper closes with future work and conclusions.

2 Background

There has been extensive work on join algorithms and techniques [4]. In this work, we only examine equi-join algorithms. We compare the traditional sort-merge join with dynamic hybrid hash join [2], hash-merge join [6], and progressive merge join [3]. The last two algorithms are designed to produce output tuples as early as possible. An overview of the join methods with respect to their CPU, memory, and I/O requirements on the client and the servers is below. Assume the two relations to be joined are R and S with $B(R)$ and $B(S)$ blocks respectively where $B(R) < B(S)$.

Sort-merge join (abbreviated MJ) requires the data sources to provide the tuples in sorted order on the join key. In many cases, the relation may already be sorted if the join is on a primary key, but in general the source may have to sort the relation. If an external sort is required, the cost is $3*B(R)+3*B(S)$. However, if R and S are stored at different sites, the sort can be done in parallel. The major advantage of sort-merge join is that the client needs a very small buffer (enough only to handle all tuples with the same join value) and never needs to store the entire relations locally. This is especially important for joins with low selectivity. No I/Os are performed at the client and the algorithm cost is dominated by network transmission time

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

and CPU time during the merge phase.

For all other join algorithms studied, the server provides the tuples in the order they are physically stored at the site. Thus, the CPU and I/O costs for the server are minimal.

Dynamic hash join [2] (abbreviated DHJ) is a variant of hybrid hash join [1] with features of adaptive hash join [9]. DHJ is not designed to produce tuples early, but does offer extremely good performance in all memory settings. It is especially resilient to small and fluctuating memory sizes. A two-pass DHJ with client memory M can join a relation of size up to M^2 and performs approximately $(2-2M/B(R)) * (B(R)+B(S))$ I/Os (not counting the initial reads from the sources).

Hash-Merge join [6] (abbreviated HMJ) is a improvement to XJoin [7] and double Pipelined Hash Join [5]. It is designed to produce results earlier and produce results even while both sources are blocked by using the downtime to join flushed partitions. The basic idea is instead of partitioning the build input first then probing, two hash tables, one for each input, are stored in memory. When a tuple arrives it is inserted into the hash table for its input and then used to probe the other hash table. This allows results to be produced faster, but also makes it necessary to keep track of what results have already been produced and handle memory overflows intelligently. HMJ improves on the earlier algorithms by performing adaptive flushing (flushing the same partition from both inputs to preserve memory balance) and by sorting flushed partitions and then later joining them using progressive merge join. The authors do not provide fixed bounds on the number of I/Os performed by the algorithm. Note that by dividing the memory between two relations decreases the maximum size that can be joined using a two-pass algorithm. Also, the complexity of performance tuning is considerably harder than DHJ.

Progressive Merge Join [3] (abbreviated PMJ) is a non-blocking sort-based join algorithm for production of early results. The idea is to sort both relations at once and then perform a join of the two in-memory partitions (using a sweep-based merge algorithm). This process is repeated until the partitioning is complete. Then, each partition is joined as with sort-merge join. The number of I/Os is exactly the same as sort-merge join, $2*(B(R)+B(S))$, except they are all performed on the client. The CPU cost is higher because the client must sort and partition both relations and perform the merge step for each partition while in memory initially and in the final merge stage. One would expect that PMJ would be inherently slower than MJ because the client must perform all the processing.

The issue to be investigated is how varying client memory and computational power affects the performance of the various join algorithms. It is obvious that all algorithms except sort-merge require the client to perform the vast majority of the work to compute the join. A major constraint is that the mediator must buffer the entire source relations. Thus, at the minimum, the mediator must have enough disk space to store both relations, and, in practice, should have sufficient memory to allow the join to be computed efficiently. The issue of forcing the client to do most of the work is even more important in N-way joins such as in [8]. We are investigating the performance of N-way hash [8], N-way progressive merge join, and N-way sort-merge join, but the results are not discussed in this paper.

Our assumption is that most sources are relational, which allows the mediator to request the data in either unsorted or sorted order. If the data is not sorted at the source, it must be sorted first before being transmitted to the mediator. Note that due to this assumption we are not treating the sources as streams, but as finite relations provided by each source.

3 Experimental Results

We performed detailed experiments on the previously listed join algorithms. The algorithms were implemented in Java and run on a Sun JVM. The system is structured as a client/server architecture. The join code was run on the client which initiated socket connections to the sources. A server provided the relations in either sorted or unsorted form depending on the client request. If the data is unsorted and is requested to be returned in sorted order, the server sorts the data using an external sort (quicksort variant). The data is streamed to the server from the client (the client does not have to request each record be delivered). To mimic slower networks, a delay period may be inserted between record transmissions.

The server machine is an Intel Pentium IV 2.26 GHz with two 60 GB drives in RAID 1 configuration. Both relations were delivered from this single server. The client machines vary from a fast client (AMD 1.6 GHz) on a 100 Mbps LAN to a slow client (900 MHz Athlon PC) connecting to the Internet via a cable modem. The join consists of two tables of 500,000 tuples of size 200 bytes each. The join key is an integer and the rest of the tuple is padded with a character field. The join is of the type primary key to primary key, and the result size is 500,000 records. We have also investigated join types of primary key to foreign key (1-N), primary key to primary key (with lower selectivity), and M-N joins with similar results. Each join configuration was run 5 times and the performance measures were averaged to come up with the final result. The variance between runs was very small. For all join algorithms, the standard deviation was less than 5% of the average time. Client memory sizes are quoted as the percentage of a *single* relation. Thus, a client memory size of 10% would hold 50,000 tuples. The server had a fixed memory size of 100,000 tuples, which required a two-pass sort to return a relation in sorted order (relevant only for MJ).

The first experiment serves as a baseline to eliminate network issues. The mediator (join client) and the server software are run on each client acting as its own server (localhost). The time for scanning a relation is 11 seconds for the fast client and 21 seconds for the slow client. The times for the slow and fast clients with memory size 10% are in Figures 1 and 2 respectively. The number of I/O operations is the same regardless of the client type (for a given memory size). For a client memory size of 10%, the number of I/O operations is given in Figure 3. Note that in this chart and all others the I/O operations shown are only for the client. All algorithms except sort-merge require no I/O operations at the server except for reading the relation.

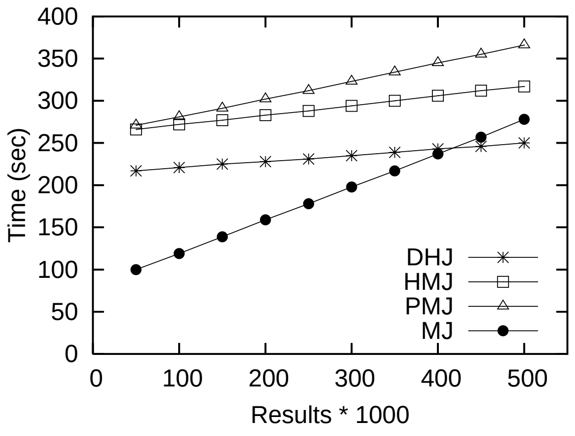

Figure 1: Times for localhost for Slow Client

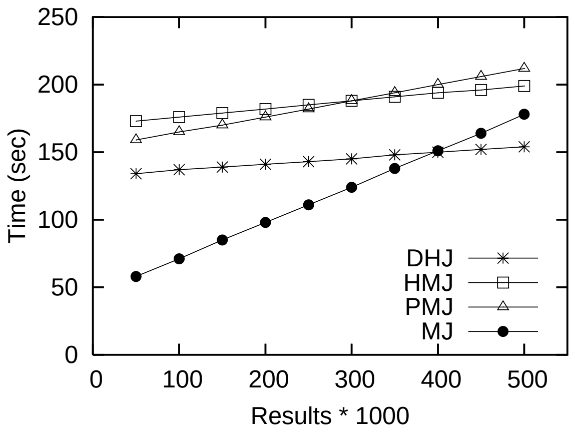

Figure 2: Times for localhost for Fast Client

The next set of experiments used the server to provide both of the input relations and the client performed the join locally. The results can be seen in Figure 4 and Figure 5.

The memory size of 10% was chosen as it has been used to evaluate HMJ and PMJ [6] and is a somewhat realistic memory size for thin, web clients. However, memory sizes as low as 1% are possible. The following figures display the number of I/Os (Figure 8) and time on the slow (Figure 6) and fast (Figure 7) client.

Since the results demonstrated that MJ and DHJ outperformed their early output counterparts, PMJ and HMJ, even when considering how fast the first tuples are produced, we also experimented with larger memory sizes, 50% (Figure 9) and 90% (Figures 10 and 11). The results are shown for the fast client only as the slow client was similar. Note that

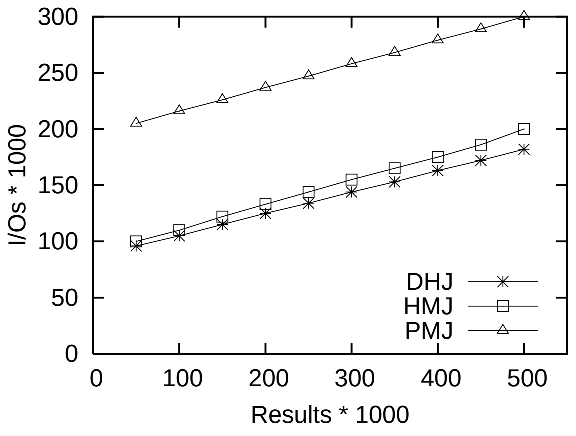

Figure 3: Client I/Os for 10% Memory

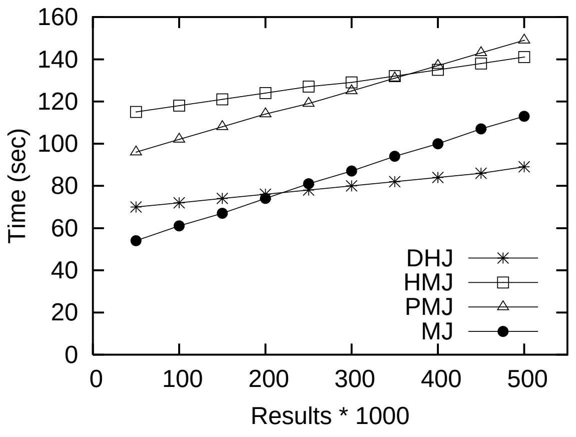

Figure 4: Times for 10% Memory Fast Client

here we begin to see PMJ and HMJ outperform for producing early results (as expected), although their overall times are still slower.

3.1 Discussion

Although we expected merge-join (MJ) to be a valuable algorithm for clients with small memory because it did not have to buffer the relations, we were somewhat surprised by its overall performance both in total time and for production of early results. This good performance can be explained by the fact that the server performs the heavy work of sorting, and the client must only perform the merge algorithm. The performance may be even better had the sorting occurred in parallel on two servers instead of one. The fact that MJ produces results early can be intuitively seen by the fact that it can start producing join tuples very soon after the first few tuples are read because of the sorted order. In comparison, the hash based algorithms get the tuples in random order and it may take a while to get tuples with matching keys. In fact, after an approximately 30 second delay (for sorting), MJ quickly produces output tuples.

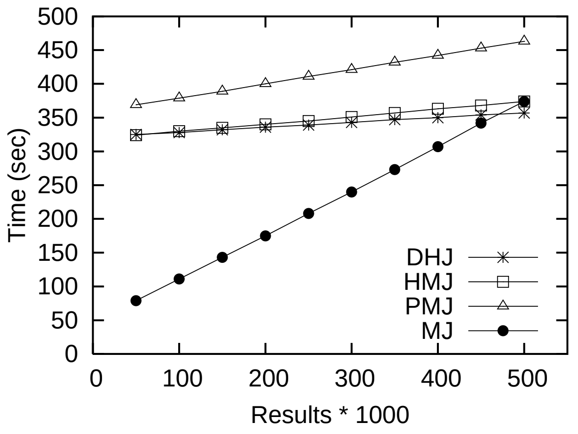

Figure 5: Times for 10% Memory Slow Client

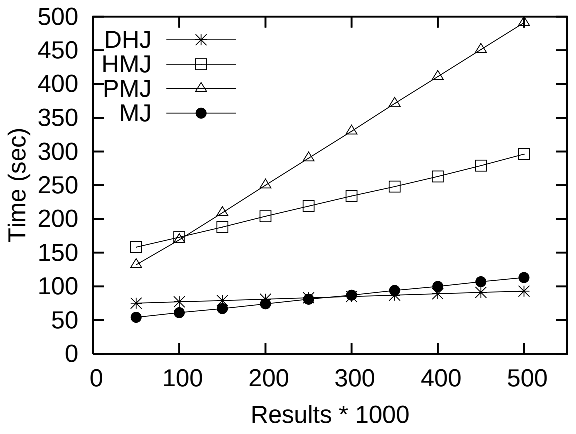

Figure 7: Times for 1% Memory Fast Client

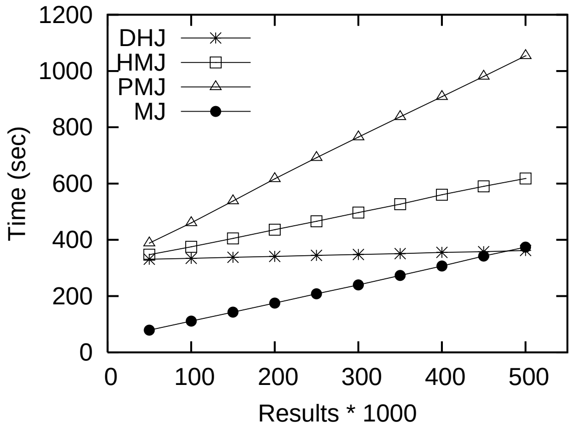

Figure 6: Times for 1% Memory Slow Client

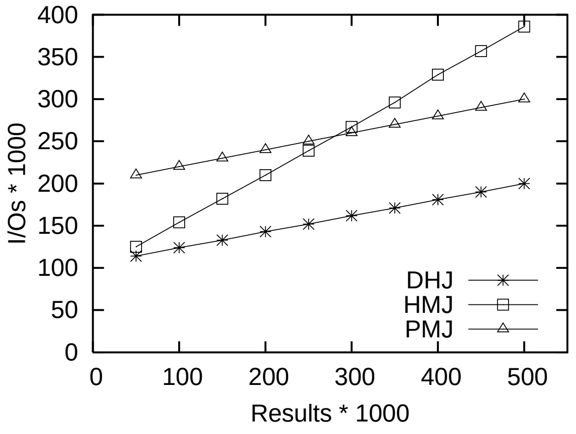

Figure 8: Client I/Os for 1% Memory

The performance of dynamic hash join (DHJ) was also better than expected. DHJ produced results quickly and had the lowest absolute times. The quick production of results is especially interesting given that no results are produced until the entire build relation is partitioned. DHJ makes up for this initial delay by very rapidly joining the probe table. Some of its speed can be accounted for by its predictable read pattern (read all of the build relation first, then all of the probe relation) which allowed for excellent buffering. This predictable nature allowed DHJ to catch MJ by the end as MJ was still accessing the server sites. The last phase of DHJ which joins disk-resident partitions on the client is extremely fast. DHJ was also incredibly resistant to low client memory and took advantage of increased memory when available.

In comparison, the two early join methods, progressive merge join (PMJ) and hash-merge join (HMJ) did not produce tuples as early as expected (except for the larger memory sizes of 50% and 90%). The reason is that for small memory sizes, the number of tuples that get produced in the first phases of these algorithms is very small. For instance with PMJ and 10% memory buffer, only 5% of each relation is processed during each partition pass. The number of matching tuples in each 5% partition is so small that most of the partitioning has to be complete before a significant number of output tuples are generated. A similar reasoning applies to HMJ as buffering 5% of the relation in memory decreases the probability that there will be many matches in memory when a given tuple is processed. The result is that these algorithms should probably not be used for small memory sizes. However, even though MJ and DHJ produce the first 20,000+ tuples faster than PMJ and HMJ, PMJ and HMJ produce the first 10,000 tuples faster than their blocking counterparts. For instance, the time to produce the first 10,000 tuples for the fast client with 10% memory is in Figure 12.

It is important to determine how many tuples the user may require before deciding on PMJ/HMJ over MJ/DHJ. Future work should consider when PMJ/HMJ should be used instead of MJ/DHJ based on the application/user requirements, and determine where the memory cutoff point occurs when the join memory available becomes too small

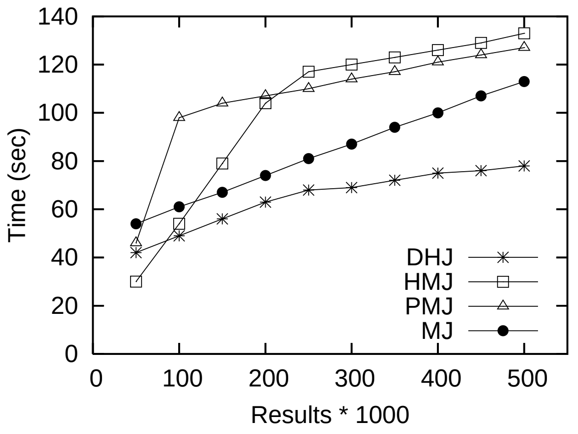

Figure 9: Times for 50% Memory Fast Client

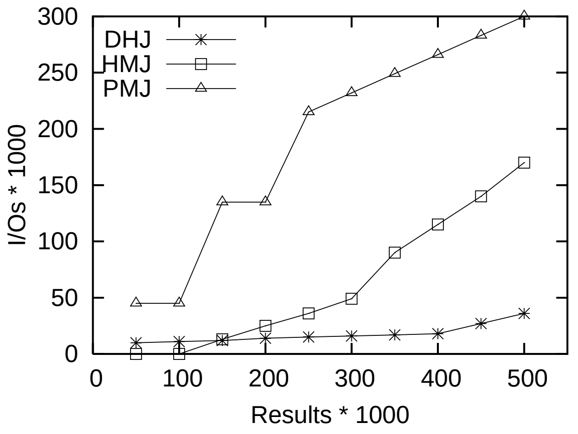

Figure 11: Client I/Os for 90% Memory

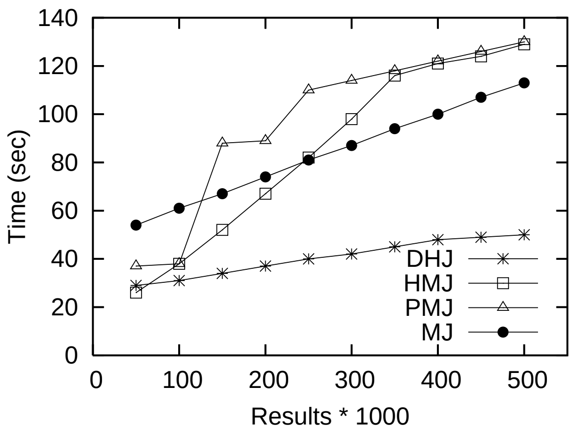

Figure 10: Times for 90% Memory Fast Client

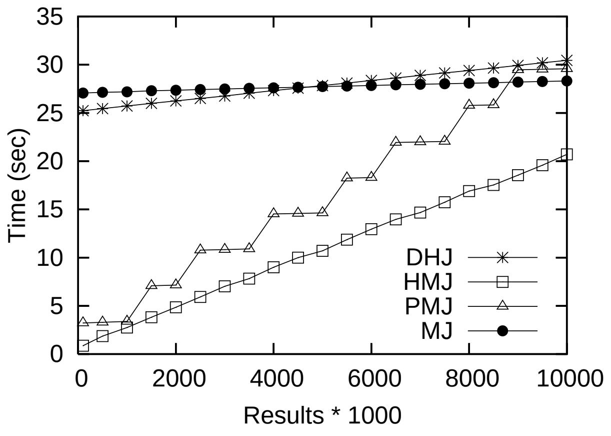

Figure 12: First 10,000 Tuples for Fast Client

to make the early production of results valuable. This cutoff point is a function of the *relative* CPU speed and available memory of the client and the servers and the *relative* size of the relations and the client memory. Although "thin clients" are constantly becoming more powerful, the size of the relations processed is increasing as well. Thus, although the absolute value of the cutoff point will vary with data size and technology improvements, determining the cutoff with given parameters is important for database optimizers as it can have a large effect on performance. The absolute performance of PMJ and HMJ significantly decreases as the memory available decreases. The overhead of producing results early has limited benefit for a small client memory and decreases the overall performance. The performance of HMJ relative to the other algorithms would be improved if source blocking was considered.

All algorithms had decreased performance on the slower client. PMJ and MJ are more affected by the decreased CPU speed. DHJ is the most resistant to a slower network, which we attribute to its predictable read pattern. One would expect that this would be even more pronounced for slower dial-up networks. We are currently conducting more experiments on this issue and attempting to determine if the performance of MJ and PMJ can be improved by different source access and buffering techniques. We are also experimenting with the *N*-way algorithms (sorting and hashing) to investigate their relative performance in this environment and modifying the experiment to access and join relational databases using JDBC.

4 Conclusion

The promise of wide-scale, dynamic, web integration requires efficient join methods for integrating results from numerous clients. In this work, we evaluated several of the proposed join methods with a specific focus on methods suitable for thin web clients with limited CPU power and memory. The sort-merge algorithm was examined as it has the benefit of requiring no fixed client memory and no buffering of source relations at the client. The performance of sort-merge join was as expected, but unexpectedly, sort-merge and dynamic hybrid hash join produce results faster than hash-merge join and progressive merge join (after the first 20,000 results). Thus, it is not always clear if the optimizer should select HMJ/PMJ over MJ/DHJ for early production of results as it depends on the number of results required and the memory size and CPU power of the client performing the join.

We are continuing our performance experiments and investigating techniques to improve the performance of current algorithms for thin client integration scenarios. Interesting future work is determining a model for predicting the relative performance of PMJ/HMJ versus MJ/DHJ given the client, server, and join relation characteristics. Such a model would be valuable for mediator query optimizers.

5 Acknowledgment

We would like to thank Mohamed Mokbel for providing the C++ code for hash-merge join.

References

[1] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *ACM SIGMOD*, pages 1–8, 1984.

[2] D. DeWitt and J. Naughton. Dynamic Memory Hybrid Hash Join. Technical report, University of Wisconsin, 1995.

[3] J.-P. Dittrich, B. Seeger, D. Taylor, and P. Widmayer. Progressive Merge Join: A Generic and Non-blocking Sort-based Join Algorithm. In *VLDB*, pages 299–310, 2002.

[4] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[5] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An Adaptive Query Execution System for Data Integration. *SIGMOD Record*, 28(2):299–310, June 1999.

[6] M. Mokbel, M. Lu, and W. Aref. Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results. In *ICDE*, pages 251–263, 2004.

[7] T. Urhan and M. Franklin. XJoin: A Reactively Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[8] S. Viglas, J. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, pages 285–296, 2003.

[9] H. Zeller and J. Gray. An Adaptive Hash Join Algorithm for Multiuser Environments. In *VLDB*, pages 186–197, 1990.