On Producing Join Results Early

[PREPRINT DRAFT: Not For Distribution]

Jens-Peter Dittrich Bernhard Seeger
Department of Mathematics and Computer Science
University of Marburg
Marburg, Germany
jens@jensdittrich.de
seeger@mathematik.uni-marburg.de

David Scot Taylor* Peter Widmayer
Institute of Theoretical Computer Science
ETH Zürich
Zürich, Switzerland
taylor@cs.sjsu.edu
widmayer@inf.ethz.ch

ABSTRACT

Support for exploratory interaction with databases in applications such as data mining requires that the first few results of an operation be available as quickly as possible. We study the algorithmic side of what can and what cannot be achieved for processing join operations. We develop strategies that modify the strict two-phase processing of the sort-merge paradigm, intermingling join steps with selected merge phases of the sort. We propose an algorithm that produces early join results for a broad class of join problems, including many not addressed well by hash-based algorithms. Our algorithm has no significant increase in the number of I/O operations needed to complete the join compared to standard sort-merge algorithms.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs between Complexity Measures

General Terms

Query Processing, Data Mining, Spatial Data

Keywords

Join Processing, Non-Blocking

1. INTRODUCTION

We study the problem of performing a join operation in a database while producing result tuples as early as possible. This will allow for piping output elsewhere (to give another process a head start) or for estimates of total output size, which may help lead to earlier abortions of joins which appear to be delivering undesired results, e.g. too many or too few result tuples.

A commonly used algorithm for joins is the SORT MERGE JOIN: based on the join attribute, each of the sets is sorted (such as by external mergesort, Knuth [13], Graefe [9]), and then both sets are joined by a merge operation that requires only a single I/O pass over each set, under the assumption that tuples with matching attributes for the join (i.e. equal for an equi join) fit in main memory at once. Here, the efficiency goal is to minimize the total number of I/O operations.

While several studies analyze the overall efficiency of join algorithms, the efficiency in terms of early result production has — to the best of the authors' knowledge — never been considered analytically. Our algorithm produces join results early (progressively) without sacrificing overall I/O-efficiency. We present the first analysis to quantify the tradeoff between I/O-efficiency and early join result production for sort-merge joins.

The results presented in this work hold for a large class of joins based on sorting, such as equi joins, spatial joins (plane sweep), temporal joins, band joins, similarity joins and more.

1.1 Previous Work

Here, we give an overview of related work on join processing with a focus on techniques that deal with early result creation. Graefe [9] contains an excellent general overview of join techniques.

Sort-Merge Joins vs. Hash-Joins

For equi joins the hash-join was found to be more efficient than the sort-merge join (Graefe [10]). For non-equi joins, however, many state-of-the-art algorithms are based on sort-merge. For *spatial intersection joins*, the algorithms of [19, 1, 5] are extensions of the sort-merge join. Arge et al. [1] shows the sort-merge join to be more efficient than a hash-based method (Patel and DeWitt [21]), even without exploiting the join-during merge technique proposed by Negri and Pelagatti [18] (see Section 1.1). The most efficient algorithms for *similarity joins* are also sort based [20, 15, 3, 6].

*Currently: Department of Computer Science, San José State University, San Jose, CA 95110

PODS 2003, June 9-12, 2003, San Diego, CA.

Even for equi joins, there exist important cases when sort-merge joins are more efficient: for multiple joins, when multiple sort-merge join operators are combined to run in a pipeline (Selinger et al. [23]), consequent operators can exploit the "interesting ordering" established by a single sorting operation.

Additionally, Li, Gao and Snodgrass [16] recently explored techniques to increase efficiency of the traditional sort-merge join in the presence of high *intrinsic skew*, which is known to adversely effect hash-based algorithms. Their experiments show that these new sort-merge join variants are much more efficient in the presence of skew than traditional sort-merge join for equi joins found in current commercial database systems.

Hashing Based Algorithms

Wilschut and Apers [25] present the Symmetric Hash-Join (SHJ) for pipelined processing of equi joins. A similar idea has been proposed by Raschid and Su [22]. For each data set SHJ builds a hash-table simultaneously in main memory. Whenever a tuple arrives, it is first inserted into its corresponding hash-table and then probed against the other. No assumption is made about the arrival frequency of tuples. Even for the case that one input is temporary blocked, the other can deliver tuples that allow the continuous production of result tuples.

The most serious limitation of SHJ is that two hash-tables have to be kept in main memory. Obviously, this requirement can not be met with very large data sets. Urhan and Franklin [24] propose XJoin, a multi-threaded extension of SHJ that can keep the hash-tables in secondary memory. A similar approach is presented by Ives et al. [12] where the algorithm is used for data integration of different active sources.

Haas and Hellerstein [11] address online aggregation when the input is received from a join. In order to produce accurate results quickly, they introduce Ripple Joins. The basic idea is to control the join processing using quality measures of the approximated aggregate value.

Luo, Naughton and Ellmann [17] recently proposed a non-blocking parallel spatial join algorithm. To the best of the authors' knowledge, this is the only work focusing on the early production of results for joins other than equi join. Their work combines techniques of SHJ, the Partition Based Spatial-Merge Join of Patel and DeWitt [21], and a modified version of the reference point method of Dittrich and Seeger [5].

Sorting Based Algorithms

Sort-based joins have previously been considered blocking operators, where first results are produced only after a considerable portion of the total runtime. This is particularly true for the original SORT MERGE JOIN of Blasgen and Eswaran [2], where both inputs are entirely sorted before being merged.

SORT MERGE JOIN has been investigated for a large number of special circumstances (Graefe [9]), such as when one of the sets is small enough to fit in main memory, or when the set sizes differ substantially (Graefe [8]). Here, we consider the problem when both sets are too large to fit in main memory.

Negri and Pelagatti [18] observed that the sorted lists may be an unnecessary byproduct of the SORT MERGE JOIN procedure. In this case, it is easy to slightly reduce the total number of I/O operations. For each of the two input sequences, the tree of external merge operations for mergesort should be modified so that for the root level, only half of the fan-in is needed. Next, the final level of each of these sorting procedures should be replaced with a single "virtual merge" in which the lists from both sets are loaded into memory, but instead of outputting two sorted lists, the lists are stepped through in linear time, outputting only the successful join tuples. As in [18], this final operation is called JOIN-DURING-MERGE. In fact, Negri and Pellagatti call their entire algorithm JOIN-DURING-MERGE, but we will refer to it as BUDGETED SORT MERGE JOIN, for reasons explained in Section 2.3. The number of I/O operations saved in this approach varies, and depends on the fan-in of the sort, and the completeness of the leaf level of the original sort trees. In the most common case, when only one merge node is needed for the two mergesorts combined, this approach eliminates one read and one write of all the data. This optimal case maximizes savings. For multi-level mergesort operations, the diminished fan-in of the mergesort root reduces the I/O operations saved, but in most cases the savings are comparable to the optimal case.

In [7] we presented a generic non-blocking technique to produce early join results. This is achieved by intermingling the sorting steps with tuple comparisons across both sets, without significantly increasing total runtime. [7] presents a basic algorithm, shows how to apply our technique to a large class of different join operations and examines a special case variant of the algorithm presented here. A series of experiments with different data sets show the efficiency of our technique.

1.2 Contributions

We examine the algorithmic side of progressively reporting result tuples as the SORT MERGE JOIN proceeds, instead of waiting for the sorting process to complete. Our PMSJ algorithm is an extension and improvement over that of [7], which focuses on experimental results. PMSJ draws its improved efficiency from an intricate interleaving of tuple comparisons for sort and for join. For instance, to further increase the rate at which results are reported, we create imbalanced external mergesort trees, with different fan-in values at different parts of the tree. The novel way we do this balances the need for immediate results against total runtime, simultaneously achieving near best-known values for each. We define a framework for measuring the overall and progressive performance of our algorithm against the currently best known algorithms. Using this framework, we provide full analysis of our algorithm, the first such analysis of a non-blocking SORT MERGE JOIN approach.

1.3 Outline

In Section 2, we make the ultimate goal of our work technically precise. In Section 3 we present our PROGRESSIVE MERGESORT JOIN (PMSJ) algorithm and its analysis. It closely matches the I/O efficiency of Blasgen and Eswaran's original SORT MERGE JOIN algorithm [2]. We further discuss practical and implementation details in Section 4. In particular, we show variants to produce all final results in sorted order, and to reduce the number of I/O operations to almost exactly match those of Negri and Pelagatti's BUDGETED SORT MERGE JOIN. We conclude in Section 5.

2. PRELIMINARIES

We begin by presenting the general framework, terminology, and variable definitions for our algorithm. Next, we propose some related problems, which serve not only to further motivate the PMSJ problem, but also to give us an "ideal" standard against which we will formally compare PMSJ.

2.1 Elementary Calculations

We consider two large sets of data, R and S. For simplicity, we will assume that they are equal sized sets of N elements each, though this is not required. Records are read B elements per page, and let $n = \lceil N/B \rceil$ be the total number of I/O operations to read (or write) all of the data from one of the sets once. For main memory of size M (in input items), and $m = \lfloor M/B \rfloor$ the number of pages which fit in main memory, the fan-in F of an external mergesort can be as large as as $m - O(1)$. Higher fan-in tends to lead to a smaller number of passes over the data, i.e., a shallower mergesort tree, and hence shorter runtime. While full main memory might be used for the evaluation of the leaf nodes of the sort tree induced by external memory mergesort, we use F to denote the fan-in, allowing the option to use a smaller fan-in[1]. To simplify notation, we may assume B divides larger values from here forward.

For the standard external memory mergesort of N items, initial runs of size M[2] can be created internally. There will be $\lceil N/M \rceil$ such runs, which constitute the leaf nodes of the external mergesort tree, and in total they will require n reads and n writes for their creation. With fan-in F, there will be $\lfloor \log_F \lceil \frac{N}{M} \rceil \rfloor$ complete levels of merging within the sort tree. Each of these levels will need to read and write all of the input. (For a tree with a single merge node, we let $F = \lceil \frac{N}{M} \rceil$.) Further, there will be one incomplete level of merges with $\lceil (\lceil \frac{N}{M} \rceil - F^{\lfloor \log_F \lceil N/M \rceil \rfloor})/(F-1) \rceil$ merge nodes of full F fan-in, each of which will use Fm read and write operations. Finally, if the above do not account for all of the leaf nodes, there will be one additional merge node, with fan-in $< F$. (There may also be one incomplete leaf node, with input size $< M$.) This assumes that the mergesort tree is constructed from the root towards the leaves, such that all levels of the tree are full except perhaps for the lowest.

This is precise but cumbersome. To simplify notation during informal discussion, we will allow "fractional" values in our calculations for the levels within the mergesort. Instead of calculating the exact number of I/O operations needed for the final, incomplete level of an F-way mergesort, we approximate the full process as needing $(\log_F \lceil \frac{N}{M} \rceil + 1)n$ reads, and writes. We use more precise values in our formal analysis (Section 3.1).

In the naive SORT MERGE JOIN algorithm, the sort goes through $\log_F \lceil \frac{N}{M} \rceil$ merge levels, plus the initial run creation for each set, and performs a final read of all data for the join step. If Z is the number of output items, and $z = \lceil Z/B \rceil$ is the number of pages of output, it takes $(2 \log_F \lceil \frac{N}{M} \rceil + 4)n$ reads, and $(2 \log_F \lceil \frac{N}{M} \rceil + 2)n + z$ writes. Between the two sets, the BUDGETED SORT MERGE JOIN will save up to $2n$ reads and writes each[3], but will not produce the two sorted sets.

2.2 Growing Sample Sizes

Before running a join operation on two huge data sets, it may be desirable to sample each to see what the output will look like, or to approximate the size of the output[4] (e.g. to estimate the similarity parameter in a similarity join). For given samples $R' \subset R$ and $S' \subset S$, $|R'| = |S'| = X$, the join problem for R' and S' mimics the original join problem with different size sets. When the subsets to join are given with X items each, the join of those items can be computed fastest by merely running the best algorithm known for the usual join problem, for instance, BUDGETED SORT MERGE JOIN from Negri and Pelagatti [18]. If we do this for a small subset, the join completes rapidly, and we get the first few join results quickly.

One important issue is how to choose a good sample size X. For the purpose of getting an expectation[5] of how many join result tuples we generate on the fly, we assume that within each set, the data is "uniformly distributed" in the following sense: the probability of a successful join operation between an item from each set is independent of the location of the items from within their respective sets. This condition is implied by stronger conditions such as requiring that the first X items from each set represent a uniform random sample from that set. Our condition implies that if there are Z join results within the entire $R \times S$ join, then subsets of size X are expected to deliver $Z(X/N)^2$ results. For $X < N/\sqrt{Z}$, less than one result is expected. For $Z = O(N)$, a relatively large sample will be needed to find interesting results, and the smaller Z is, the larger the sample must be. The complicating factor is that Z is not known; Z is needed to choose a good sampling size, yet it is estimated by the sample.

The best we can hope for is a dynamic, growing sample, produced by an algorithm which delivers join results for increasing sized subsets and nevertheless completes in the best known time. This allows sample sizes to be taken in a fully adaptive way, in the sense that no choices for the sample size X (or a priori estimates of Z) are needed. We propose PMSJ, which comes close to this behavior: after T I/O operations, for any T, the number of joins reported is close to the number that would be produced from a sample, if the sample size was chosen to run in T operations. That is, we are competitive with the above ideal standard.

---

[1] Smaller fan-in values may result in more levels for the mergesort tree, but Graefe reports that trade-offs in seek, latency, and transfer times, in conjunction with buffering and forecasting techniques, usually combine to make relatively small fan-in sizes optimal. See [9], Section 4.2.

[2] If the initial sorting is done via replacement-selection (see Knuth [13]), the initial runs are of expected size $2M$, resulting in only half as many leaf nodes for the mergesort. It can be added to any mergesort based algorithm (including ours). See Graefe [9] for discussion.

[3] In the optimal (and most common) case, when the new tree uses only a single merge node, the savings will be $4n$ total I/O operations. In general, multi-level trees save $\frac{F-2}{F-1}2n$ to $(4 - O(1)/F)n$ I/O operations.

[4] A uniform sample of the inputs does not give a uniform sample of the outputs. For in-depth discussion of the difficulty of efficiently and accurately sampling the *output* of a query, see Chaudhuri, Motwani, and Narasayya [4]. The input samples can still be used to estimate the total output size. See also [7].

[5] We aim at expected case behavior. Worst-case results to obtain even one result tuple for a join takes asymptotically just as long as sorting, by reduction from external element uniqueness.

2.3 Generalized Budgeted Sort Merge Join

Let us turn the above reasoning around: given an I/O budget, how can one maximize the total number of successful joins found from R and S? If we have a fixed budget of I/O operations, it is reasonable to pick as large a sample from each as can be run through the algorithm with the fastest completion. Of course, not knowing how many results will come from the join makes precise budgeting of the I/O impossible. In joining two subsets of size X, $Z(X/N)^2$ result tuples are expected, at an I/O budget of approximately $\lceil\frac{X}{B}\rceil(4\log_F 2\lceil\frac{X}{M}\rceil + 2)$ (ignoring the output of join result tuples). (A more precise I/O budget and full analysis is given in Lemma 7, but our goal here is to introduce intuitive ideas.)

Note that it might be impossible to progressively produce this number of results using this number of I/O operations as X grows. For a single, given X value, however, we can run BUDGETED SORT MERGE JOIN to get the expected join result tuples. Thus, our GENERAL BUDGETED SORT MERGE JOIN performance represents a whole family of algorithms, parameterized by the sample size X of items per list. Any sample size corresponds to a budget of I/O operations, and an instance with this budget produces no output at all until the root node of GENERAL BUDGETED SORT MERGE JOIN is reached, and during this (linear time) JOIN-DURING-MERGE, all of the output will quickly be produced. The speed at which join result tuples are produced within that final JOIN-DURING-MERGE node depends on how large the samples are: the larger X is, the more rapidly the tuples will come once the final node is reached (though it will take longer to reach that node).

Our ultimate goal is to create a single algorithm which progressively increases the sample size, and for each sample size compares with the best performance (measured in join result tuples versus I/O operations), thus measuring it against the entire family of GENERAL BUDGETED SORT MERGE JOIN algorithms with specific I/O budgets. As one run of our algorithm uses more and more I/O operations, it is progressively compared to a GENERAL BUDGETED SORT MERGE JOIN algorithm with an I/O budget to match. We will measure our algorithm's deviation from this ideal curve in two ways. The first (*delay*) measures how many extra I/O operations our algorithm has vs. GENERAL BUDGETED SORT MERGE JOIN. The second (*output efficiency*) measures how many results we progressively produce compared to those BUDGETED SORT MERGE JOIN could produce using the same I/O, that is, we compare our results and I/O performance to that of the entire GENERAL BUDGETED SORT MERGE JOIN family.

It will become clear that our PMSJ algorithm allows many ways to balance output efficiency against delay. For this common trade-off between two goals — greedy (produce join results immediately) vs. long range (find all join results as quickly as possible) — we introduce an interesting technique for our mergesort tree evaluation which nearly optimizes both simultaneously.

One fact is painfully obvious even if we could match the entire GENERAL BUDGETED SORT MERGE JOIN performance curve with one algorithm: the number of joins reported at the beginning of a run is a small percentage of the whole. If the sets are large enough to require several levels within the mergesort, this seems to be inherent to the problem.

3. PROGRESSIVE MERGESORT JOIN

In this section we introduce our non-blocking join algorithm PROGRESSIVE MERGESORT JOIN (PMSJ). Ideally, we would like to create a *single* algorithm which will progressively produce results which match the performance of the entire GENERAL BUDGETED SORT MERGE JOIN family as closely as possible, rather than just matching it at one point. In order to match the total I/O performance, no more than $(4\log_F\lceil\frac{N}{M}\rceil + 2)n + z$ I/O operations may be used to run the algorithm.

Our approach is to interleave join operators within the mergesort procedure. We extend our work from [7] in several ways: we carefully order the evaluation of the mergesort tree, strategically place just a few join operators, and perform merges much more frequently in some parts of the tree. This new algorithm will produce many more early join results. We then quantify our efficiency using the measures introduced in Section 2.3.

We treat sets R and S symmetrically. We first present an internal "level" (Figure 1) within the tree of the sorting process of PMSJ, and then will describe the top and bottom of the tree. In the figure, each node represents a process which takes input streams from its children processes, merges them, and outputs a stream with longer sorted subsequences (runs). First, consider the solidly drawn nodes at the top of the figure. Data from R will be divided equally between these nodes for R, and the single, leftmost JOIN-DURING-MERGE node for R at the same level. The same holds for S, using the nodes drawn with dotted lines. (To simplify our description and analysis, we are assuming here that N is a size which will make the bottom level of our tree complete. More formal analysis is given in Section 3.1.) The nodes at the bottom of the figure represent nodes the next full level down, for which corresponding statements hold. The tree will be evaluated by post-order traversal, starting with the JOIN-DURING-MERGE nodes on the left. Once we finish evaluating such a node, its siblings are evaluated (any arbitrary post-order will do), and then its parent (another JOIN-DURING-MERGE node) is evaluated.

For a node-by-node description, we begin with the nodes on the right side of the tree. They are very much like the external mergesort nodes in a standard SORT MERGE JOIN algorithm. Each node takes F sorted subsequences from its children and merges them, producing a longer sorted subsequence. For each of these nodes, either all of the children streams are from R, or they are all from S, and all of these nodes have full fan-in F. The R and S children nodes are drawn in an alternating way, to show a "uniform progression" through the two data sets, but many possible evaluation orders (including strict left-to-right post-order) give the same performance.

The nodes on the left side of the tree are JOIN-DURING-MERGE nodes, similar to the root node of the BUDGETED SORT MERGE JOIN algorithm, but each is modified to produce sorted outputs for each R and S subset. They have different fan-in values, and take streams of both R and S values. They merge their sorted subsets from R (S) into longer sorted subsets from R (S), and while they do this, produce any join results between the two subsets, while they are in memory. Care must be taken to not re-report previous join results, but this is straightforward: the values from the leftmost substream of R have already been compared to those in the leftmost substream from S, while no other

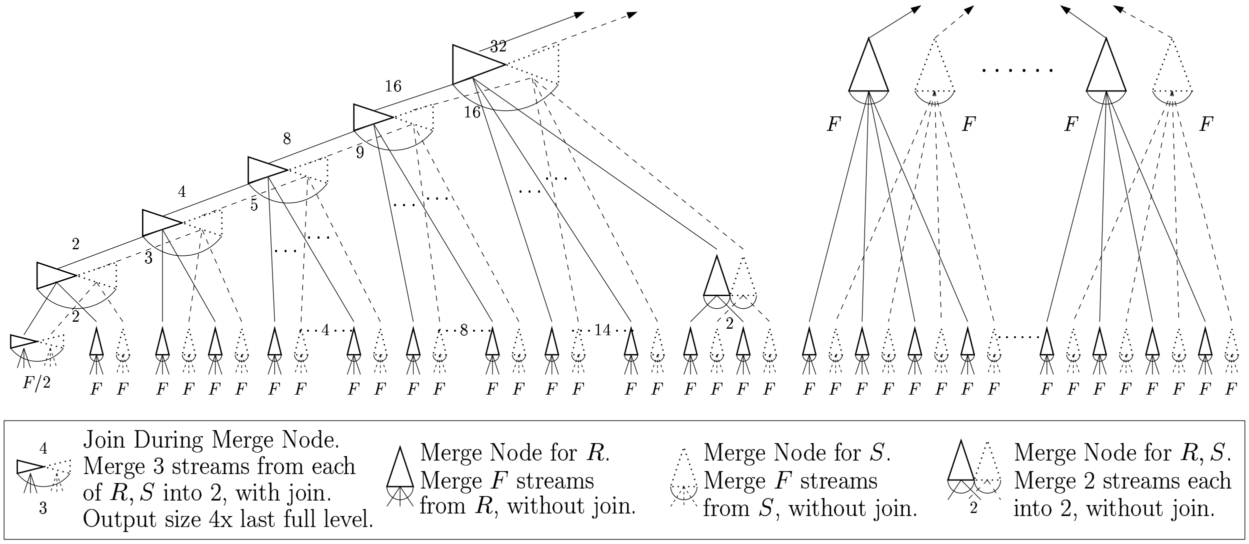

Figure 1: One full level of Progressive Mergesort Join to the next ($F = 64$).

join results have been reported. See [7] for implementation details, and Section 4.3 for a variant without duplication elimination.

Notice that the left side of the tree has many more levels than the right side. We consider the right side to define the levels of the tree, while the left side of the tree has $\log_2 F - 1$ extra *intermediate* levels per full level, with many different fan-in values. To simplify later analysis, we have assumed that F is a power of 2.

The last type of node in the tree (there is only one such node per level) has two children from each of R and S. It merges these two pairs into two sorted subsets. The node is only added so that its parent node will have total fan-in F (vs. $F+2$), a technical detail which is often swept aside using the standard assumption that there are always $O(1)$ extra pages available in memory. More importantly, these nodes can also be used to eliminate a more bothersome problem: within the algorithm, if a value is repeated nearly M times (or if there are many "similar" values in a similarity search), it can occur that there is not enough room in the memory to hold all of these values, and still maintain fan-in F. (Li, Gao, and Snodgrass [16] also addresses the problem of heavy skew.) We can use one of these nodes before each Join-during-Merge node, and convert all Join-during-Merge nodes into regular join nodes with fan-in two (one each from R and S). With this conversion, there will be enough space in memory to hold all values, unless there are more than M repeats. While this conversion will add to the overhead of the algorithm, this same problem must be addressed by any join algorithm.

It remains to describe the top and bottom of the tree. The top of the tree looks similar to the left side of Figure 1, with only the single Join-during-Merge node as the root, and that root does not need to produce the sorted subsets of R and S. The bottom of the tree will look similar to the bottom of two external mergesort trees (one for each of R and S), except for the two leftmost leaf nodes. Other than these, there will be $\frac{N}{M}-1$ leaves from each of R and S, size M each, and they will each sort their contents within internal memory and output the sorted list. The leftmost leaf node (which will be the first node in the tree to evaluate) takes $M/2$ records from each of R and S, sorts the two subsets individually, and then while they are in memory, performs a join on them (see also Footnote 2). It produces the sorted subsets and the join results as output. The second "leaf" node (and the second node overall to be evaluated) is similar, except that it also takes the sorted subsets from the first leaf as inputs (needing only 2 more pages), reports only new join results, and outputs a sorted subset of length M for each of R and S. To simplify later analysis, this second "leaf" (it does have a child, but also looks like other leaves because it also performs an internal sort on raw, unsorted data) will be considered to be on the bottom full level of the tree, with all of the "regular" leaves, while the very first leaf evaluated will be treated as a unique subleaf.

In this modified tree, the leftmost leaf gets merged more frequently than the rightmost. In total, they are merged $\log_2 F \log_F \lceil \frac{N}{M} \rceil + 1$ and $\log_F \lceil \frac{N}{M} \rceil$ times respectively. The fan-in of the leftmost nodes are determined by balancing the need to produce results immediately, while not wanting to delay future levels by too much. They also allow any nodes within the same (full) level to have the same number of children as descendants.

3.1 Analysis

We want to analyze PMSJ in two ways: how many I/O operations does it use, and how quickly does it produce results along the way. Let the level of the root node be 0, and each full level node has a level 1 larger than the full level node above it. We make several assumptions to simplify calculations: B divides $M/2$, $N = MF^i$ for some integer i (so $\log_F \lceil \frac{N}{M} \rceil = \log_F \frac{N}{M}$ is an integer), and $F = 2^j$ for some integer j. (If these assumptions do not hold, similar results, with different constant terms, will follow.)

Lemma 1. *(a) All regular leaves are associated with size M subsets of R or S. The full level leaf* Join-during-Merge *node is associated with a size M subset of R and S.*

(b) Any regular non-leaf node is associated with sets F times larger than each of its F children.

(c) The R and S subsets associated with any full level JOIN-DURING-MERGE *node are (each) the same size as the subsets associated with regular nodes at the same level. Any* JOIN-DURING-MERGE *node is associated with sets twice as large as its child* JOIN-DURING-MERGE *node.*

PROOF. (a) The regular leaves have size M by definition. The very first node evaluated takes size $M/2$ subsets from each of R and S. Its parent, the bottom full level JOIN-DURING-MERGE node, also takes size $M/2$ subsets from each, and merges these results with those from the first node, getting size M subsets from each.

(b) Every regular parent node merges the results from each of its children, which are one full level down in the tree.

(c) This holds at the bottom level of the tree by (a). Intermediate level JOIN-DURING-MERGE nodes are arranged to have enough regular children nodes to match the size of the sets of their child JOIN-DURING-MERGE node: first one R and S node are needed, then two, four, etc. Thus, each JOIN-DURING-MERGE node doubles the size of its child JOIN-DURING-MERGE node. After $\log_2 F - 1$ such intermediate levels, the next (full level) JOIN-DURING-MERGE node doubles the size again, bringing that node to have F times larger subsets of R and S than the JOIN-DURING-MERGE node one full level down. These sets will again match in size with the regular nodes on the same level by (b). $\square$

LEMMA 2. *The tree contains $\log_F \frac{N}{M}$ full merge levels. Level $\log_F \frac{N}{M}$ contains the leaf nodes.*

PROOF. The leaves are associated with size M subsets from R or S. Going up i full level merges, the nodes will be associated with size MF^i subsets. When $i = \log_F \frac{N}{M}$, $MF^i = N$, which matches the size of the root node sets. The root is level 0, so the lowest merge level is level $\log_F \frac{N}{M} - 1$. $\square$

LEMMA 3. *(a) At full level i, for $i \leq \log_F \frac{N}{M}$, there are $F^i - 1$ regular nodes are associated with size $MF^{\log_F \frac{N}{M} - i} = N/F^i$ subsets of R, and the same number associated with S. A single* JOIN-DURING-MERGE *node is associated with size N/F^i subsets of R* and *S.*

(b) For $0 < j < \log_2 F$, the jth intermediate JOIN-DURING-MERGE *node above full level i is associated with size $2^j N/F^i$ subsets from R* and *S.*

PROOF. (a) This holds immediately from Lemmas 1 and 2. (b) This holds by (a) and from the proof of Lemma 1.c. $\square$

LEMMA 4. *After the evaluation of a regular node at level i, the total number of I/O operations (excluding join results) used in the evaluation of that node's entire subtree is $(\log_F \frac{N}{M} - i + 1)2n/F^i$.*

PROOF. After level i, the N/F^i (Lemma 3.a) data associated with a regular node has undergone $\log_F \frac{N}{M} - i$ merges (Lemma 2). For each merge level within the node's subtree, all of the data is read and written, at n/F^i reads and writes each. All data of the subtree is also read and written within the leaves of the subtree. $\square$

LEMMA 5. *(a) During the evaluation of a* JOIN-DURING-MERGE *node at full level i, for $i > 0$, $4n/F^i$ I/O operations take place (excluding join results).*

(b) During the evaluation of the jth intermediate level JOIN-DURING-MERGE *node following level i, $2^j 4n/F^i$ I/O operations take place (excluding join results).*

(c) The "bookkeeping node" which evaluates just before the JOIN-DURING-MERGE *node at level i uses $8n/F^{i+1}$ I/O operations.*

PROOF. (a) The JOIN-DURING-MERGE node is associated with size N/F^i sized sets from each of R and S (Lemma 3.a), and all of this data is read once, merged (no I/O), and output in two sorted lists.

(b) This holds as in (a), except that the intermediate level JOIN-DURING-MERGE nodes are associated with size $2^j N/F^i$ subsets from each of R and S (Lemma 3.b).

(c) The node has 4 children nodes (two from R, two from S) of size N/F^{i+1} each, and all data is read, merged (no I/O), and written. $\square$

LEMMA 6. *Let $T[i]$ be the total number of I/O operations (excluding join results) used to evaluate the entire subtree rooted at the* JOIN-DURING-MERGE *node at full level i.*

(a) $T[\log_F \frac{N}{M}] = 6m = \frac{6n}{F^{\log_F \frac{N}{M}}}$

(b) For $0 < i < \log_F \frac{N}{M}$:

$$T[i] = n\left(\frac{4(\log_F \frac{N}{M} - i + 2)}{F^i} + \sum_{k=i+1}^{\log_F \frac{N}{M} - 1} \frac{4}{F^k} + \frac{2}{F^{\log_F \frac{N}{M}}}\right)$$

PROOF. (a) The very first node evaluated (the subleaf) reads two sets of $M/2$ unsorted data, and writes two sets of M sorted data, for $2m$ total I/O operations. The second node evaluated, the first full-level JOIN-DURING-MERGE node, does the same, and also reads and writes the data from the first node, for $4m$ I/O operations, and $6m$ total in this subtree.

(b) Within the subtree, we will sum the I/O operations for the subtrees rooted one full level down, the root node, the intermediate level JOIN-DURING-MERGE nodes evaluated since the last full level nodes, and the last bookkeeping node. We will use induction on the subtree rooted at the JOIN-DURING-MERGE node one level down, which uses $T[i+1]$ I/O operations. The tree has $2(F-1)$ regular subtrees (split between type R and S) one full level down (which can be seen indirectly by Lemmas 1.c and 3), and to evaluate each uses $(\log_F \frac{N}{M} - (i+1) + 1)2n/F^{i+1}$ I/O operations (Lemma 4), for $4n(F-1)(\log_F \frac{N}{M} - i)/F^{i+1}$ total. By Lemma 5, the numbers of I/O operations used by the root node, the intermediate level JOIN-DURING-MERGE nodes, and the bookkeeping node are $4n/F^i$, $\sum_{k=1}^{\log_2 F - 1} 2^k 4n/F^{i+1} = (F-2)4n/F^{i+1}$, and $8n/F^{i+1}$ respectively. Summing these 3 terms gives $8n/F^i$. With the subtrees, this gives the recursive equation:

$$T[i] = T[i+1] + \frac{4n(F-1)(\log_F \frac{N}{M} - i)}{F^{i+1}} + \frac{8n}{F^i}$$

Proof by induction follows. For the base case, the recursive equation with $T[\log_F \frac{N}{M}] = 6m$ gives $T[\log_F \frac{N}{M} - 1] = 12Fm + 2m$. $\square$

THEOREM 1. *The total number of I/O operations used in* PMSJ *is*

$$n\left(4\log_F \frac{N}{M} + 6 + \sum_{k=1}^{\log_F \frac{N}{M} - 1} \frac{4}{F^k} + \frac{2}{F^{\log_F \frac{N}{M}}}\right) + z$$

$$< 4n\left(\log_F \tfrac{N}{M} + 3/2 + \tfrac{1}{F-1}\right) + z$$

PROOF. The fundamental difference between the full PMSJ analysis and a subtree rooted at the JOIN-DURING-MERGE node at level i is that the root does not output the sorted "subsets" of R and S. Thus, it uses $2n$ fewer write operations than a subtree of the same size would. Plugging $i = 0$ into the equation in Lemma 6.b, and subtracting $2n$ gives the initial equation. The rewrite is an upperbound on the telescoped summation terms. □

We can now compare our total I/O operations against those of BUDGETED SORT MERGE JOIN: in BUDGETED SORT MERGE JOIN, there are $\log_F \frac{N}{M}$ merge levels (this allows BUDGETED SORT MERGE JOIN to use double fan-in on the top level, giving it a slight advantage, but otherwise the size of N would allow a full bottom level in our algorithm but not theirs). Each merge level reads in all of the data ($2n$ reads) and all but the root level write all of the data ($2n$ writes). The leaf level also reads and writes all of the data, for $n(4\log_F \frac{N}{M} + 2) + z$ I/O operations. This nearly matches the I/O operations for PMSJ, if we allow for one extra read and write of all the data ($4n$ I/O operations). (We note that without the bookkeeping node at each level, the recursion gives less than $n(4\log_F \frac{N}{M}+6)$ I/O operations. To implement this would require fan-in $F + 2$ at each full level JOIN-DURING-MERGE node, or similarly, increasing the fan-in from $F/2+2$ to $F/2+4$ at the previous JOIN-DURING-MERGE node.)

In order to accurately compare how quickly our algorithm generates join tuples compared to GENERAL BUDGETED SORT MERGE JOIN, we need to be more precise with I/O calculations for BUDGETED SORT MERGE JOIN than we have been previously.

LEMMA 7. *Let X be divisible by M.* BUDGETED SORT MERGE JOIN *uses $4\frac{X}{B}(\lfloor\log_F \frac{X}{M}\rfloor + 3/2) - 2mF^{\lfloor\log_F \frac{X}{M}\rfloor}$ or more I/O operations to join two sets of size X, excluding those used for join results.*

PROOF. For integer j, k and $0 \le j < \log_2 F$ and $1 \le \alpha < 2$, let $X = \alpha 2^j F^k M/2$. If $j = 0$ and $\alpha = 1$, the algorithm needs $4\frac{X}{B}(\lfloor\log_F \frac{2X}{M}\rfloor + 1/2)$ and the inequality holds, so assume that $j > 0$ or $\alpha > 1$. There will be $\frac{2X}{M}$ leaves in the tree (size M each), which are spread between two levels. At most $F^{\lfloor\log_F \frac{2X}{M}\rfloor}$ can be on the higher of the two levels, and each of these will undergo $\lfloor\log_F \frac{2X}{M}\rfloor$ merges. The data in these leaves is read and written at the leaves, and within every merge level (without a write at the root), which will use $2mF^{\lfloor\log_F \frac{2X}{M}\rfloor}(\lfloor\log_F \frac{2X}{M}\rfloor + 1/2)$ I/O operations. The bottom level will contain at least $\frac{2X}{M} - F^{\lfloor\log_F \frac{2X}{M}\rfloor}$ leaves, and these will undergo one extra level of merges, for $2m(\frac{2X}{M} - F^{\lfloor\log_F \frac{2X}{M}\rfloor})(\lfloor\log_F \frac{2X}{M}\rfloor + 3/2)$ I/O operations. Summing these two and cancelling terms, we get $4\frac{X}{B}(\lfloor\log_F \frac{2X}{M}\rfloor + 3/2) - 2mF^{\lfloor\log_F \frac{2X}{M}\rfloor}$, at least as large as the number in the lemma. □

LEMMA 8. *Let $X = 2^l M \le N$ for some positive integer l. To join X elements from each set,* PMSJ *needs at most $4\frac{X}{B}(\lfloor\log_F \frac{X}{M}\rfloor + 3) - 2mF^{\lfloor\log_F \frac{X}{M}\rfloor}$ I/O operations, excluding those used for join results.*

PROOF. Define integers i, j such that for $0 \le j < \log_2 F$, $X = 2^j \frac{N}{F^i} (= 2^j F^{\log_F \frac{N}{M} - i} M)$. To analyze X elements, PMSJ must complete j JOIN-DURING-MERGE nodes following full level i. (If $j = 0$, it must just complete the full level JOIN-DURING-MERGE node.) Summing the I/O operations needed to perform the subtrees rooted at the full level ($T[i] + 2(2^j - 1)(\log_F \frac{N}{M} - i + 1)2n/F^i$ by Lemma 4) and any intermediate level JOIN-DURING-MERGE nodes ($\sum_{k=1}^{j} 2^k 4n/F^i = (2^j - 1)8n/F^i$ by Lemma 5.b), a total of $T[i] + 4n(2^j - 1)(\log_F \frac{N}{M} - i + 3)/F^i$ I/O operations are used. Filling in $T[i]$ from Lemma 6 and telescoping the summation terms, this is fewer than $4\frac{2^j n}{F^i}(\log_F \frac{N}{M} - i + 3) - \frac{4n}{F^i} + \frac{4n}{F^i(F-1)}$. Combining the last terms and making replacements ($\frac{2^j n}{F^i} = \frac{X}{B}$ and $\lfloor\log_F \frac{X}{M}\rfloor = \log_F \frac{N}{M} - i$) gives at least $4\frac{X}{B}(\lfloor\log_F \frac{X}{M}\rfloor + 3 - \frac{F-2}{2^j(F-1)})$. Finally, $X/2^j B = mF^{\lfloor\log_F \frac{X}{M}\rfloor}$, and we assume that $F > 2$, so the inequality holds.

Comparing Lemma 8 to 7, the extra I/O operations come from $2X/B$ writes for last JOIN-DURING-MERGE node (which are piped to the next level here, but not in BUDGETED SORT MERGE JOIN), and from the additional JOIN-DURING-MERGE nodes near the top of the tree, which do not use their full fan-in, causing about $2X/B$ extra reads and writes each. □

LEMMA 9. *After joining size X subsets from R and S, $Z(X/N)^2$ results are expected.*

PROOF. Any join result from the X^2 possibilities will be reported. By assumption, the probability of a successful join for each is Z/N^2. □

THEOREM 2. *Suppose that in its progression,* PMSJ *has used T I/O operations, and has joined size X subsets from R and S. If* BUDGETED SORT MERGE JOIN *is run on a sample size chosen to use T I/O operations, let* $\text{BestCase}[X]$ *and* $\text{WorstCase}[X]$ *be the best and worst number of results expected from it, as a ratio, compared to those produced by* PMSJ. *(T excludes join results).*

$$(a)\ \text{BestCase}[X] \le \left(\frac{\lfloor\log_F \frac{X}{M}\rfloor + 3}{\lfloor\log_F \frac{X}{M}\rfloor + 3/2}\right)^2$$

$$(b)\ \text{WorstCase}[X] \le 4\left(\frac{\lfloor\log_F \frac{X}{M}\rfloor + 3}{\lfloor\log_F \frac{X}{M}\rfloor + 3/2}\right)^2$$

PROOF. (a) The best performance for PMSJ comes just after it has completed a JOIN-DURING-MERGE node. Let that node have size X. At this time, PMSJ has used at most the number of I/O operations in Lemma 8. Assuming that the other algorithm has processed sets of size αX, we plug into Lemma 7. For $\alpha \ge (\lfloor\log_F \frac{X}{M}\rfloor + 3)/(\lfloor\log_F \frac{X}{M}\rfloor + 3/2)$, it will have more I/O operations than PMSJ. We can see from Lemma 9 that squaring this result gives the comparative number of results expected.

(b) Let X be such that $4\frac{X}{B}(\lfloor\log_F \frac{X}{M}\rfloor + 3) - 2mF^{\lfloor\log_F \frac{X}{M}\rfloor} = T$. If a JOIN-DURING-MERGE node has just completed, we are done by (a). Otherwise, consider X' to be the size of the sets of the next JOIN-DURING-MERGE node to complete if the algorithm were to proceed. After that node, performance would be $\text{BestCase}[X']$. Instead, the last JOIN-DURING-MERGE node to have completed had size $X'/2$ (Lemma 1.c), and it must have produced $1/4$ of the output expected from the size X' sets (Lemma 9). Allowing GENERAL BUDGETED SORT MERGE JOIN the number of I/O operations PMSJ needs to process X' sized sets, yet only allowing PMSJ to process size $X'/2$ sets gives the result. □

If performance is only measured at the completion (or the start) of nodes from within PMSJ, the worst case performance improves slightly: the numerator term of $\lfloor \log_F \frac{X}{M} \rfloor + 3$ can be replaced by $\lfloor \log_F \frac{X}{M} \rfloor + 2$. In this case, the worst time to measure will be just before the evaluation of a JOIN-DURING-MERGE node, while above it may actually take place *during* the evaluation of a JOIN-DURING-MERGE node.

COROLLARY 1. *For any fixed I/O budget $T > m$,* PMSJ *and* GENERAL BUDGETED SORT MERGE JOIN *(only the latter with T specified) are expected to produce the same order of results.*

4. VARIANTS AND PRACTICAL ISSUES

We view our final approach as a balance between the greedy and long term goals. Such a balance is often better than either extreme: once some results are in hand, it is easier to be a bit more patient until the next "big payoff" (the join at the next level in the tree). In our PMSJ algorithm, the intermediate level JOIN-DURING-MERGE nodes work towards producing immediate results. (Rather than being lazy or eager, these merges might be classified as over-eager, as they are performed with a lower fan-in than the other merge nodes. BUDGETED SORT MERGE JOIN is at the other extreme (long term), trying to complete the entire process as early as possible.

4.1 Less Uniform Variants

Of the many variants possible, it is useful to consider variants which are not uniform throughout the different levels of the tree. For instance, if the goal is to optimize the worst ratio, our scheme can be improved upon: notice that for small X values, Theorem 2 implies an output efficiency of 1/4 down to 1/16 (or to just under 1/7 if we don't measure performance between nodes), while for very large X values, it ranges from just under 1 to just under 1/4. Allowing a small sacrifice to efficiency of large samples, the worst-case ratio for small X can be slightly augmented, by using different structure for the lower levels of the tree.

In another example of how non-uniform behavior might be useful, notice that most of the overall I/O delay comes from the intermediate level nodes just under the root-level merge, because these nodes process large portions of R and S. When users reach these nodes, they should have better estimates as to how many total join results will be produced, and thus may be committed to running the algorithm to completion. If the intermediate level nodes near the root of the tree are eliminated, nearly all of the I/O overhead also disappears. (To simplify implementation, this will work best with separate sample and result streams as discussed in Section 4.3.) For instance, getting rid of the intermediate level nodes between level 0 and 1 will reduce the total I/O's to under $n(4 \log_F \frac{N}{M} + 2 + 4/(F-1)) + z$, only increased from those of BUDGETED SORT MERGE JOIN by the $4n/(F-1)$ term.

4.2 Single Level Trees

While the algorithms of Sections 3 and 4.1 have good performance for multilevel sort trees, many external mergesort processes have only one merge level. (See also Footnote 1 for cases when multi-level trees may be preferred even if not required.) In this case, the I/O overhead of our procedure may be larger than practical. For trees which only require one merge node with greatly reduced fan-in ($\leq F/2$), BUDGETED SORT MERGE JOIN uses only $4n$ reads and $2n + z$ writes[6].

In this case, we propose a simplified version of our PMSJ algorithm. We modify BUDGETED SORT MERGE JOIN, replacing all leaf nodes to look like our very first leaf node: each will share memory between R and S, and produce join results between the already loaded sublists. (This will result in $2\lceil N/M \rceil$ total leaves, each with a sorted set output for R and S.) No intermediate level JOIN-DURING-MERGE nodes are added. Just as in BUDGETED SORT MERGE JOIN, this tree will have $4n$ reads, and $2n + z$ writes[7], for a total I/O which matches that of BUDGETED SORT MERGE JOIN. At the beginning of the procedure, results will be produced at a rate to match the GENERAL BUDGETED SORT MERGE JOIN performance, but instead of accelerating, they continue to be produced at that same rate until the final merge node. Once that final node is reached, remaining results will be produced quickly. Implementation details and experimental results of this simplified, one-level version are given in [7].

4.3 Sorted Results

For some applications, it is useful to give the result tuples in sorted order. PMSJ can be easily modified to give final results in sorted order if allowed 2 streams of output: an unsorted sample stream, and the final sorted output stream. The final JOIN-DURING-MERGE node can be modified to output all successful joins to the output stream (without eliminating formerly reported results by checking which input streams the results come from), and it will output all results in sorted order. The earlier reported results can be piped to a sample stream, used to estimate the total number of results expected, and to see some examples. If the sample stream allows for a few repeated results, the algorithm runs faster due to simplified code. (The "duplicate check" code can be eliminated, speeding runtime without changing the I/O operations.) The total number of results projected can be adjusted to still be accurate. Further, because each merge node should return many more results than all of its descendants combined, most results in the sample will still only be reported once.

5. CONCLUSIONS

We have presented an algorithm which uses a new technique to progressively produce join tuples. After T I/O operations, for any $T > m$, it is expected to produce the same number of results which could be produced by the best SORT MERGE JOIN algorithm, to within an $O(1)$ multiple. This holds even though T is not specified to PMSJ and it is specified to the optimal SORT MERGE JOIN algorithm. Our technique centered on the idea of concentrating extra join operations along the "spine" of the external mergesort tree, which is small when compared to the whole tree. We plan to apply this approach, which holds some similarity to that

[6]The merge node may also include unsorted subsets of R and S which are not in any leaf node, decreasing the reads and writes by up to $m - \lfloor |S|/m \rfloor - \lfloor |R|/m \rfloor$ each. For large data sets, this will be a small savings. See also the Graefe [9] discussion on eager vs. lazy merging.

[7]As in Footnote 6, a marginal decrease of the number of write operations may be possible. Here, there is only an $m - \lfloor 2|S|/m \rfloor - \lfloor 2|R|/m \rfloor$ decrease.

taken in Depth-First Iterative-Deepening Search (Korf [14]), to other problems for which multi-level trees are prevalent.

There is another benefit of our method, different from those discussed thus far. Traditional processing techniques for joins are particularly unacceptable when joins are processed on data items that are delivered from remote sources in an unpredictable network. Non-blocking join algorithms like XJoin (Urhan and Franklin [24]) suggest to activate intermediate steps in case of blocking input. These intermediate steps are based on data that is already kept temporarily on disk and does not depend on the availability of data items from the sources. The advanced processing strategies of PMSJ fit very well to such a scenario: with half of the data from each source, it should produce one fourth of the results (the maximum possible with this amount of data).

6. ACKNOWLEDGMENTS

We thank the anonymous referees for their helpful suggestions.

7. REFERENCES

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable Sweeping-Based Spatial Join. In *International Conference on Very Large Data Bases*, pages 570–581, 1998.

[2] M. W. Blasgen and K. P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):362–377, 1977.

[3] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon Grid Order: An algorithm for the similarity join on massive high-dimensional data. In *ACM SIGMOD International Conference on Management of Data*, pages 379–388, 2001.

[4] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *ACM SIGMOD International Conference on Management of Data*, pages 263–274, 1999.

[5] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *International Conference on Data Engineering*, pages 535–546, 2000.

[6] J.-P. Dittrich and B. Seeger. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *ACM SIGKDD International Converence on Knowledge Discover and Data Mining*, pages 47–56, 2001.

[7] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive Merge Join: A generic and non-blocking sort-based join algorithm. In *International Conference on Very Large Data Bases*, pages 299–310, 2002.

[8] G. Graefe. Heap-Filter Merge Join: A new algorithm for joining medium-size inputs. *IEEE Transactions on Software Engineering*, 17(9):979–982, 1991.

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

[10] G. Graefe. Sort-Merge-Join: An idea whose time has(h) passed? In *International Conference on Data Engineering*, pages 406–417, 1994.

[11] P. J. Haas and J. M. Hellerstein. Ripple Joins for online aggregation. In *ACM SIGMOD International Conference on Management of Data*, pages 287–298, 1999.

[12] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD International Conference on Management of Data*, pages 299–310, 1999.

[13] D. Knuth. *The Art of Computer Programming, Volume III: Searching and Sorting*. Addison Wesley, second edition, 1998.

[14] R. E. Korf. Depth-First Iterative-Deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):35–77, 1985.

[15] R. A. Kyuseok Shim, Ramakrishnan Srikant. High-dimensional similarity joins. In *International Conference on Data Engineering*, pages 301–313, 1997.

[16] W. Li, D. Gao, and R. T. Snodgrass. Skew handling techniques in sort-merge join. In *ACM SIGMOD International Conference on Management of Data*, pages 169–180, 2002.

[17] G. Luo, J. F. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. In *International Conference on Data Engineering*, pages 697–705, 2002.

[18] M. Negri and G. Pelagatti. Join During Merge: An improved sort based algorithm. *Information Processing Letters*, 21(1):11–16, 1985.

[19] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *ACM SIGMOD International Conference on Management of Data*, pages 326–336, 1986.

[20] J. A. Orenstein. An algorithm for computing the overlay of k–dimensional spaces. In *International Symposium on Advances in Spacial Databases*, pages 381–400, 1991.

[21] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1996.

[22] L. Raschid and S. Y. W. Su. A parallel processing strategy for evaluating recursive queries. In *International Conference on Very Large Data Bases*, pages 412–419, 1986.

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[24] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *Data Engineering Bulletin*, 23(2):27–33, 2000.

[25] A. N. Wilschut and P. M. G. Apers. Pipelining in query execution. In *Conference on Databases, Parallel Architectures and their Applications*, pages 68–77, 1991.