Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm*

Jens-Peter Dittrich Bernhard Seeger
Department of Mathematics and CS
University of Marburg
Marburg, Germany

{dittrich,seeger}@mathematik.uni-marburg.de

David Scot Taylor Peter Widmayer
Department of CS
ETHZ
Zürich, Switzerland

{taylor,widmayer}@inf.ethz.ch

ABSTRACT

Many state-of-the-art join-techniques require the input relations to be almost fully sorted before the actual join processing starts. Thus, these techniques start producing first results only after a considerable time period has passed. This blocking behavior is a serious problem when consequent operators have to stop processing, in order to wait for first results of the join. Furthermore, this behavior is not acceptable if the result of the join is visualized or/and requires user interaction. These are typical scenarios for data mining applications. The ‚off-time' of existing techniques even increases with growing problem sizes.

In this paper, we propose a generic technique called Progressive Merge Join (PMJ) that eliminates the blocking behavior of sort-based join algorithms. The basic idea behind PMJ is to have the join produce results, as early as the external mergesort generates initial runs. Hence, it is possible for PMJ to return first results very early. This paper provides the basic algorithms and the generic framework of PMJ, as well as use-cases for different types of joins. Moreover, we provide a generic online selectivity estimator with probabilistic quality guarantees. For similarity joins in particular, first non-blocking join algorithms are derived from applying PMJ to the state-of-the-art techniques.

We have implemented PMJ as part of an object-relational cursor algebra. A set of experiments shows that a substantial amount of results are produced, even before the input relations would have been sorted. We observed only a moderate increase in the total runtime compared to the blocking counterparts.

1. INTRODUCTION

Operations on large databases may take a long time to complete. This is true in particular for join operations on disk-based data, where the number of both, input data as well as results, can be extremely large. In many settings, it is desirable to deliver at least a few result items as fast as possible. The reason may be that all results need further processing, which should start as early as possible. This behavior is highly desired for operations such as online aggregation [13, 11], or when a quick visual inspection of first results is desired, in order to decide whether the join operation should continue or be aborted.

The sort-merge join [4] is a popular algorithm for joining two sets of data items. It first sorts both sets and then steps in a merge-like fashion through both sequences. This technique requires a linear order on the input sets, related to the join predicate. All commercial DBMS provide an implementation of the equi-join based on the sort-merge paradigm. For many joins other than equi-join, like spatial and similarity joins, the state-of-the-art algorithms [1, 8] are based on the sort-merge paradigm.

The most serious problem of the sort-merge join is that first results are delivered only after the input sets have been read entirely and (at least partially) sorted. Sort-merge is therefore blamed for being a *blocking* algorithm that prevents the fast delivery of results. The same problem occurs in all algorithms based on the sort-merge paradigm, particularly the ones for processing joins other than equi-join. Similarity joins in the context of data mining are an excellent example. They are essential for basic operations like clustering [5] and outlier detection. The most efficient techniques for similarity joins [6, 8, 17, 23, 27] are sort-based and therefore return answers only after a long processing period. This prevents user-interaction which is mandatory for online data mining. The same observation holds for online aggregation queries [13] that receive their input from a sort-based join. The blocking behavior of the sort-merge join is illustrated in Figure 1.

Ideally, a join operator for two sequences of data items should have the following properties. First and foremost, the first few results should be delivered rapidly without much delay of the ones remaining. Moreover, these results should be sufficiently representative for the entire set such that accurate estimations of aggregates can be supported. Second, the overall efficiency of the join should not be much worse than other state-of-the-art methods based on the sort-merge paradigm. Note that there is a general conflict between the size of the bulk in which first results are produced and the speed at which the remaining are delivered. Furthermore, the number of main memory computations should be similar to those of sort-merge join.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

*This work has been supported by grant no. SE 553/2-2 from DFG.

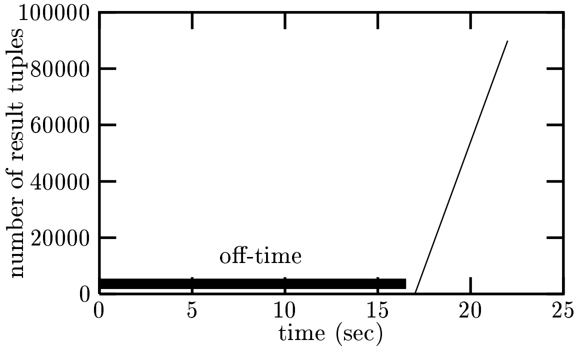


Figure 1: The blocking behavior of a sort-based join: The number of results as a function of runtime

In this paper, we present a new kind of sort-merge join called *Progressive Merge Join (PMJ)* which satisfies all of the requirements stated above. PMJ is derived from the sort-merge join. Instead of keeping the sorting and joining phases strictly separated, PMJ attempts to compute results already during the sorting phase. It does so by sorting both input sets simultaneously and by joining data items that are in main memory at the same time. Obviously, the first result item can be expected significantly earlier than the completion of the sorting. The total runtime of sorting, however, will slightly increase by a small constant factor, due to the fact that main memory must be divided between the input sets. Nevertheless, the number of main memory computations will also stay close to the one of the sort-merge join, simply because data items meet in main memory according to the mergesort process only. Moreover, an additional overhead of PMJ is that an answer can be produced more than once and therefore, an additional check for almost every answer (except the first ones) is required whether it is already produced or not. PMJ delivers the first result rapidly, but it does not necessarily deliver any desired number of result items as fast as possible. We show, theoretically as well as experimentally, that the total runtime of PMJ is close to the one of its blocking counterpart. Due to the non-blocking behavior, PMJ does not return the join results in order as it is known from the original sort-merge join. This gives us the advantage that the early results are sufficiently representative for computing an online selectivity estimator. Under the assumptions of independent input relations and a random delivery order, see also [11], the estimator gives quality guarantees independent from the underlying join predicate. If desired, a slight modification of PMJ can still produce sorted output after an initial sample.

Our algorithm is generic in the sense that it can be used whenever the sort-merge paradigm is applicable to processing joins. This paper provides conditions that have to be satisfied by a specific type of join. It also contains a proof of correctness for the generic approach, based on these conditions. The algorithm is generic enough for smooth integration into extensible database systems for easy customization [3].

The rest of this paper is organized as follows: the next section introduces our notation and discusses related work. After that we introduce our new algorithm PMJ (Section 3). In Section 4 we present results obtained from experiments where PMJ is used for implementing different kinds of joins (equi join, spatial join, similarity join).

2. PRELIMINARIES

2.1 Notation

In this section, we introduce the notation used throughout the paper. Let R and S be two data sets. Let $P_{join}(r, s)$, $r \in R$, $s \in S$ be a binary predicate. The join of R and S is given by

$$R \bowtie S \ = \ \{(r, s) \mid P_{join}(r, s),\ r \in R,\ s \in S\}\,.$$

We assume in our work that a 'useful' order can be defined on the elements of R and S (otherwise a sort-based approach would not make much sense). Given such an order, one can reduce the number of comparisons between elements of R and S by a great deal. For a set T we use T' to denote the sequence that contains all elements of T sorted w.r.t. that ordering.

2.2 Related Work

In this section we give an overview on related work on join processing with a focus on techniques that deal with early result creation. Contrary to our approach, these techniques are based on hashing.

Algorithms based on Hashing. In [30], Wilschut and Apers present the Symmetric Hash-Join (SHJ) for pipelined processing of equi-joins. A similar idea has been proposed in [26] by Raschid and Su. For each data set SHJ builds a hash-table simultaneously in main memory. Whenever a tuple arrives, it is firstly 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 limit of SHJ is that both hash-tables have to be kept in main memory. Obviously, this requirement can not be met dealing with very large data sets. In [29], Urhan and Franklin propose the XJoin, a multi-threaded extension of SHJ that can keep the hash-tables on secondary memory. XJoin partitions the data into buckets. When main memory becomes full, the largest bucket is written to disk and registered for later processing by a parallel thread. Since answers might be produced more than once, XJoin also provides an efficient duplicate removal technique based on time-stamping. A quite similar approach is presented in [14] where the algorithm is used for data integration of different active sources.

Haas and Hellerstein [11] address the problem of online aggregation when the input is received from a join. In order to produce accurate results fast, they introduce Ripple Joins. The basic idea is that the quality issues of the approximated value of the aggregate control the processing of the join. This requires an asymmetric processing of the input relations.

Recently, Luo, Naughton and Ellmann [19] 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. The paper combines the findings of SHJ and PBSM [24], adding a modified version of the reference point method [7], in order to eliminate duplicates. The algorithm partitions the problem in a top-down manner as it is known from hashing, but its partitioning function preserves the ordering.

Algorithms based on Sorting. So far, sort-based joins have always been considered blocking operators, where first results are produced only after a considerable portion of the total runtime. This is particular true for the original sort-merge join [4] where both inputs are entirely sorted at first and merged. In [20], Negri and Pelagatti present an improvement that alleviates the blocking behavior of the sort-merge join. The authors propose an online processing of the last merge operation of the *external merge sort*, i.e. instead of writing two completely sorted data sequences (one for each data set) to disk, both data sets are merged directly until p runs, $2 < p \leq M$, are left on disk. One input-page for each run is then reserved in main memory and a heap is used to determine the run that contains the next minimal element w.r.t. the ordering. By following this approach one entire write and one entire read of the data sets are saved. This technique is used in many systems nowadays. However, the blocking behavior is still visible as it is shown in Figure 1 where the results are obtained from an experiment, using this optimized version of a sort-merge join.

Despite the fact that sort-merge joins are blocking, many state-of-the-art algorithms make use of this technique. For spatial intersection joins, the algorithms presented in [22, 16, 1, 7] are true extensions of the sort-merge join. The same holds for similarity joins, where we are only aware of the sort-based algorithms proposed in ([27, 23, 17, 6, 8]). The blocking behavior of these methods is obviously in contradiction to the interactive processing style of data mining – the primary target of these algorithms.

3. PROGRESSIVE MERGE JOIN

In this section we introduce our non-blocking generic join algorithm termed *Progressive Merge Join (PMJ)*. The basic idea behind PMJ is to sort both data sets simultaneously, right from the creation of the first run, and progressively produce results from the subsets in main memory. PMJ makes use of a so called *sweep area* that can be tailor-cut to support all standard sort-based joins like band-joins [18], temporal joins [28], spatial joins [1] and similarity joins [23, 8]. Therefore, many different joins can benefit from PMJ with respect to the early production of results.

In the following, we will first present the most important algorithms of PMJ (see Section 3.1). In Section 3.2 and 3.3 we prove the correctness of PMJ. In Section 3.4 we discuss the efficiency of our approach. In Section 3.5 we propose a generic online estimator for the selectivity of joins being processed by PMJ. Finally, we present important extensions of PMJ in Section 3.6.

3.1 Algorithms

In analogy to external sorting PMJ consists of two phases described in the following. Each phase is then explored in more detail in a separate subsection.

In the first phase, PMJ starts reading as much data as possible from input sets R and S into the available main-memory. Both subsets are then sorted using an internal algorithm like Quicksort. The sorted sequences are joined using an in-memory join algorithm. After that, both sequences are temporarily written to external memory. Result tuples of the in-memory join are delivered directly to the next operator [10] and are not further considered. PMJ continues with loading subsets in memory from the remaining input, sorting and joining these subsets until the input is completely processed.

Given:
-input sets R and S
-number of items M that can be held in memory
-maximal fan-in of a merge F

Let Q be an empty set;
Let RES be the empty result set;

Phase 1 ('Join during run creation'):
1 While $(R \neq \{\} \vee S \neq \{\})$ {
2 Let $\hat{R}$ be a subset of R and $\hat{S}$ be a subset of S
3 where $|\hat{R}| + |\hat{S}| \leq M$
4 $R = R \backslash \hat{R}$; $S = S \backslash \hat{S}$;
5 Sort $\hat{R}$ into sequence $\hat{R}'$, sort $\hat{S}$ into sequence $\hat{S}'$;
6 $RES = RES \cup$ **earlyJoinInitialRuns**($\hat{R}'$, $\hat{S}'$);
7 Write $\hat{R}'$ and $\hat{S}'$ to external memory;
8 $Q = Q \cup \{(\hat{R}', \hat{S}')\}$;
9 }

Phase 2 ('Join during merge'):
10 While($|Q| > 1$) {
11 Let $\hat{Q}$ be a subset of Q, $|\hat{Q}| \leq F/2$;
12 Let $(\hat{R}', \hat{S}')$ be a tuple of two empty sequences;
13 $RES = RES \cup$ **earlyJoinMergedRuns**($\hat{Q}$, $(\hat{R}', \hat{S}')$);
14 Write $\hat{R}'$ and $\hat{S}'$ to external memory;
15 $Q = \{Q \backslash \hat{Q}\} \cup \{(\hat{R}', \hat{S}')\}$;
16 }

Figure 2: Main algorithm of PMJ

In the second phase, PMJ generates longer runs by merging the sequences that were temporarily written to external memory. Again, this is performed in a way that a merge process is active for each of the input sets. The runs obtained from merging are temporarily written to disk. One of the important ideas of PMJ is that during a merge the next results of the join are produced.

3.1.1 *Join during run creation*

Let us follow the algorithm presented in Figure 2. The algorithm starts with four input parameters: the input sets R and S, the number of items that can be held in memory (M), and the fan-in of the merge (F). In the first phase, PMJ starts by creating sorted sub-sequences ('runs'). This resembles the run creation phase of external merge sort, but in contrast, PMJ processes two data inputs at once, see lines 2&3 in Figure 2, where $|\hat{R}|+|\hat{S}| \leq M$. For sake of simplicity, let us assume $|\hat{R}| \simeq |\hat{S}| \simeq M/2$. This however is not fixed and can be modified at runtime, as required by the ripple join, for instance. Both subsets are first sorted and then, the sorted sub-sequences are joined by using an appropriate internal join algorithm (line 6). The pair ($\hat{R}'$, $\hat{S}'$) is then temporarily stored in Q on external memory (line 7). PMJ continues creating pairs of sorted sequences until all items of both input sets are processed (line 1).

The internal join of the first phase is processed by a call of `earlyJoinInitialRuns`, see Figure 3. This routine follows the ideas of the plane-sweep paradigm [25] that is also known

```
1  Function earlyJoinInitialRuns(Sequences R', S'){
2      Set RES = {};
3      SweepArea R_M = {}, S_M = {};
4      While (R' ≠ {} ∨ S' ≠ {}) {
5          r = First(R');
6          s = First(S');
7          If (S' = {} ∨ (R' ≠ {} ∧ r ≤ s)) {
8              R_M.insert(r);
9              RES = RES ∪ S_M.query(r);
10             R' = R'\{r};
11         }
12         Else {
13             S_M.insert(s);
14             RES = RES ∪ R_M.query(s);
15             S' = S'\{s};
16         }
17     }
18     return RES;
19 }
```

Figure 3: The method `earlyJoinInitialRuns`

```
1  TYPE SweepArea {
2      Predicate P_join;
3      Predicate P_rm;
4      Set X = {};
5      Procedure insert(Element v) {
6          X = X ∪ {v};
7      }
8      Function query(Element v) {
9          X = X \ {u | P_rm(u,v), u ∈ X};
10         return { (u,v) | P_join(u,v), u ∈ X};
11     }
12 }
```

Figure 4: The data type SweepArea

```
1  Function earlyJoinMergedRuns({(R'_0, S'_0),.,(R'_n, S'_n)}, (R̂', Ŝ')){
2      Set RES = {};
3      SweepArea R_M = {}, S_M = {};
4      While (∃m : R_m ≠ {} ∨ ∃S_m ≠ {}) {
5          Let r =First(R'_i), r ≤ First(R'_l) ∀ i ≠ l;
6          Let s =First(S'_j), s ≤ First(S'_l) ∀ j ≠ l;
7          If (S'_j = {} ∨ (R'_i ≠ {} ∧ r ≤ s)) {
8              R_M.insert(r);
9              RES = RES ∪
10                 (S_M.query(r) \ {(r,x)| x ∈ S_i});
11             R'_i = R'_i \ {r};
12             R̂' = R̂' ∪ {r};
13         }
14         Else {
15             S_M.insert(s);
16             RES = RES ∪
17                 (R_M.query(s) \ {(x,s)| x ∈ R_j});
18             S'_j = S'_j \ {s};
19             Ŝ' = Ŝ' ∪ {s};
20         }
21     }
22     return RES;
23 }
```

Figure 5: The method `earlyJoinMergedRuns`

from processing spatial joins [1]. In particular, it employs a specific data structure termed *sweep area*, see Figure 4, which corresponds to a set where items are efficiently managed and retrieved with respect to a given predicate. The sorted sequences are processed item by item. The smallest unprocessed item of the sequences is first inserted into the associated sweep area (see line 8&13 of Figure 3) and then used for retrieving elements from the other sweep area. The answers to these queries belong to the response set of the join. We make sure that our storage requirement for the sweep area is no larger than that of any other (standard) sort-merge join algorithm for the same predicate. We achieve this by removing elements that are not relevant for queries that are issued later, see line 9 of Figure 4. This also improves the runtime of queries.

We want to point out that the join predicate only occurs in the specific sweep area. This is actually the reason why our join is broadly applicable to different kinds of joins. For the equi-join we obtain the following settings of the predicates: The join predicate $P_{join}(u, v)$ tests whether u and v are equal or not. The remove predicate $P_{rm}(u, v)$ tests whether u is lower than v. After a query is processed, the sweep area consists of elements that are equal. A more detailed discussion will follow in Section 3.2.

3.1.2 Join during merge

In the second phase, PMJ merges sorted sub-sequences ('runs') to larger ones. This resembles the merge phases of external merge sort, but in contrast, PMJ simultaneously processes a merge for each input set. The parameter F denotes the maximal fan-in, i.e. the maximal number of sequences that may be merged by one operation. Q contains all tuples of sequences that have to be processed. PMJ chooses a subset of tuples $\hat{Q} \subseteq Q$, where $|\hat{Q}| \leq F/2$ (line 11 in Figure 2). After that, `earlyJoinMergedRuns` (see Figure 5) is called with the subset $\hat{Q}$ and a tuple $(\hat{R}', \hat{S}')$ containing two empty output sequences (line 12/13 in Figure 2). `earlyJoinMergedRuns` merges the input sequences into two larger sequences $\hat{R}', \hat{S}'$ which are then joined directly and written to disk (line 14 in Figure 2). Finally, the pair of sorted sequences $(\hat{R}', \hat{S}')$ is added to Q. The process of merging and joining sequences is repeated (line 10 in Figure 2) until Q contains only a single tuple (R', S'), i.e. two sorted sequences of the join inputs R and S.

The method `earlyJoinMergedRuns` (see Figure 5) simultaneously calculates both the merge of the sorted sequences and their join results. Note, that writing and joining sequences is not sequentially processed, as the reader may expect from our description, but simultaneously processed in order to avoid temporary buffering and output of the sequences. The function first determines one of the smallest unprocessed elements of the input sequences w.r.t. the ordering used in the join (lines 5&6). Then, the smallest element of them is chosen (line 7) and inserted in the associated sweep area. Then, a query is processed on the other sweep area (lines 9&16). The algorithm continues with moving the element from its original sequence to the output sequence. This step of joining and merging input sequences is repeated until both input sequences have been fully consumed (line 4).

Our current implementation of PMJ realizes line 10&17 in Figure 5 as follows. Instead of using one sweep area for each data set, there is a sweep area for each input run. Whenever an item r of the i^{th} run is selected (line 5-7 in Figure 5), join partners are only retrieved from the sweep areas that are not associated to the i^{th} run. This guarantees that no previously delivered results are reproduced, and it therefore gives the advantage that a result can immediately be delivered right after its production.

3.2 Completeness

In this section, we show that PMJ is generally applicable and correct. In order to prove correctness of PMJ, we first have to show that PMJ computes the entire response set (*Completeness*). We first start with the introduction of two rules that have to be satisfied by the generic algorithms that are derived from the plane-sweep paradigm.

The basic idea of the plane-sweep algorithm is that for each input set the data within a "little" window (with respect to the ordering) is kept in the sweep area. Note that as long as we do not remove data from the sweep area, the algorithm shows similarities to SHJ [30]. When the sweep area is organized as a list, we actually would perform a kind of symmetric nested-loops join [11]. It is however of utmost importance to efficiency that the remove predicate P_{rm} is applied just before issuing a query. This keeps the sweep area small and reduces the number of objects being examined in the consecutive query. In order to return all results of the join, however, the remove predicate P_{rm} and join predicate P_{join} have to satisfy the following conditions $\forall u$ and $\forall v$:

$$\forall w \geq v : P_{rm}(u,v) \Longrightarrow \neg P_{join}(u,w) \tag{1}$$

$$\forall w \leq v : \neg P_{rm}(u,v) \Longrightarrow \neg P_{rm}(u,w) \tag{2}$$

The first condition ensures that when v deletes u from the sweep area, the elements greater than v do not match with u anymore. The second condition guarantees when v does not remove u from the sweep area, all elements w that are lower equal to v do not remove u from the sweep area. Both conditions are necessary for the correctness of the join. In particular, the following Lemma holds.

LEMMA 1. *Let P_{rm} and P_{join} be predicates satisfying conditions 1 and 2. Let R' and S' be sorted sequences. Then, earlyJoinInitialRuns (see Figure 3) returns $R' \bowtie S'$.*

As a direct consequence from Lemma 1 we obtain that the original sort-merge join on R and S is correct when it is performed within our algorithmic framework. This is because the input sets are first entirely sorted and then `earlyJoinInitialRuns` is called once using (R', S') as input parameter.

THEOREM 1 (COMPLETENESS OF PMJ). *Given two input sets R, S and predicates P_{join} and P_{rm}, PMJ reports all tuples from $R \bowtie S$.*

PROOF (COMPLETENESS OF PMJ). We show that each result tuple of the original sort-merge join is indeed produced by PMJ. Let (u, v) be an answer obtained from the original sort-merge join and let us assume (without loss of generality) that $u \leq v$ holds and u was inserted into the sweep area before v. Consequently, $P_{join}(u, v)$ holds, and with condition (1), we get $\neg P_{rm}(u, v)$.

Now let us take a look at PMJ. PMJ builds up a merge-tree for each of the input sets R and S. We make sure without sacrificing efficiency that the skeletons of these merge-trees are identical, by adding dummy nodes (representing empty sets) if needed. We distinguish two cases. The first case is that u and v belong to the same initial run (leaf of the merge-trees). For such a run, `earlyJoinInitialRuns` is called. Due to Lemma 1, (u, v) is then in the response set of PMJ. The second case is that u and v do not belong to the same initial run. Then, there is a node in the merge-trees where they meet for the first time, i.e. the tuple (u, v) could not have been generated before. `earlyJoinMergedRuns`, see Figure 4, performs the merge. Because $u \leq v$, u is inserted into the sweep area (before v is inserted). Therefore, the consecutive query cannot produce the result tuple.

Next, let us consider the situation when v is inserted into the sweep area. We show that u still has to be in the sweep area and PMJ therefore reports (u, v). Since (u, v) is a result of the join, $P_{join}(u, v)$ is satisfied and with condition 1, $\neg P_{rm}(u, v)$. By using condition (2), we obtain that for all $w \in S$, $u \leq w \leq v$ $\neg P_{rm}(u, w)$. Since only these elements of S are processed between the insertion of u and v, we conclude that u has to be in the sweep area when v is processed. □

In the following, we show that the different types of join are supported within our algorithmic framework of PMJ. In Table 1, we report the predicates of the joins and their sorting order. In addition, we show the underlying data type of the join. For example, a spatial join is defined for rectangles where a rectangle is an array of intervals. The similarity join is defined for points where the join condition is based on the Euclidean distance. The sorting order as well as the remove predicate of the similarity join is based on the Z-code of a multi-dimensional point [22].

LEMMA 2. *The remove predicates of the different joins satisfy the conditions 1 and 2.*

PROOF (SPATIAL JOIN). As an example we show that the Lemma is correct for the spatial join. Let u and v be tuples. We show that the remove predicate satisfies the two rules:

1. First, let us assume $P_{rm}(u, v)$ is satisfied, i.e. v removes u from the sweep area. Then, $u.R[0].max <$ $v.R[0].\text{min}$ holds. Let w be a tuple and $w > v$. It follows that $w.R[0].\text{min} > v.R[0].min (> u.R[0].max)$. Hence, $P_{rm}(u, w)$ is true.

2. Second, let us assume $P_{rm}(u, v)$ is not satisfied and let w, $w \leq v$, be a tuple. Then, $w.R[0].min \leq v.R[0].min \leq u.R[0].max$. Consequently, $P_{rm}(u, w)$ is not satisfied.

□

3.3 Uniqueness

In this section, we show that each result of the join is reported exactly once (*Uniqueness*). The method `earlyJoinMergedRuns` of PMJ does not only produce results, but also contains commands to remove duplicates (see line 10 and 17 in Figure 5). Our specific implementation of removing duplicates as discussed in Section *3.1.2* is not relevant for the following Theorem.

THEOREM 2 (UNIQUENESS OF PMJ). *PMJ reports each answer of the join exactly once.*

Join Algorithm	Data Type	Order by	$P_{join}(u,v)$	$P_{rm}(u,v)$
equi join	attribute A	A	$u.A = v.A$	$u.A < v.A$
band join	attribute A	A	$\|u.A - v.A\| \leq \varepsilon$	$u.A < v.A - \varepsilon$
temporal join	interval $I = [min, max]$	$I.min$	$u.I \cap v.I \neq \emptyset$	$u.I.max < v.I.min$
spatial join	rectangle $R = [I_x, I_y]$	$R[0].min$	$u.R \cap v.R \neq \emptyset$	$u.R[0].max < v.R[0].min$
similarity join	point P	ZCode(P)	Euclid($u.P, v.P$)$\leq \varepsilon$	ZCode($u.P$) no prefix of ZCode($v.P$)

Table 1: The predicates and sorting order of different join algorithms

PROOF (UNIQUENESS OF PMJ). The proof is conducted by an induction on the height of the merge-tree. It is obvious that a merge-tree of height 0 does not generate duplicates (because it refers to a join of two initial runs).

Let n be an integer and let us assume that we have already proved that merge-trees of height j, $j \leq n$ do not deliver duplicates. Let us consider a merge-tree of height $n+1$. The last merge of this tree is performed by a call of `earlyJoinMergedRuns` where the input refers to runs $(R_0, \ldots, R_m, S_0, \ldots, S_m)$ that are generated by merge-trees of height n. The answers that have been computed until now are given by

$$\bigcup_{i=1}^{m} R_i \bowtie S_i.$$

These answers would be produced once more, but line 10 and line 17 remove them from the result set. □

3.4 Efficiency

In the following, we first examine the I/O cost of PMJ expressed in the number of page accesses. We do not consider the cost for reading the input and writing the output. Let B and M be the number of items that fit into a page and into main memory, respectively. Let N be the number of input items. For simplicity, we assume that M and N are multiples of B. To conveniently support two input streams and one output stream for each of both relations, we assume that $M \geq 6B$.

First, let us assume that the size of the sweep area (P) is always smaller than $M - 6B$. Then, $M - P$ items can be used for the run generation and merging phases of external merge-sort. The I/O cost of PMJ is determined by the cost of external merge-sort where each input set makes use of half the available memory. Therefore, the I/O cost of PMJ is then given by the following formula:

$$\Theta(N/B \; log_{(M-P)/B} N/B)$$

The remaining question is what does happen in case the sort-merge join requires more than the available memory. For example, for an equi-join this only happens if a certain join value occurs almost M times — a problem that can be addressed in our setting at least as well as in any other (and even better for instance in the case of spatial joins [1] by means of an appropriate sweep area organization). Let us first discuss this issue in detail for the equi-join, band-join and temporal join. For those types of joins, the queries that are processed on the sweep area return a join result for each element of the sweep area. If we modify PMJ such that a constant portion of memory larger than B is allocated for the sweep area, the total I/O cost is given by the following formula:

$$\Theta(N/B \; log_{M/B} N/B + k/B)$$

where k denotes the number of results of the join. For the spatial join we obtain the same formula by applying a technique called distribution-sweeping [1]. The authors of [1] however observed for their real data sets that the size of the sweep area did not exceed $O(\sqrt{N})$. Therefore, they conclude that the sweep area fits in memory for almost all up-to-date computers. A very similar observation was made for the similarity join [8] where the sweep areas in the experiments required only a marginal portion of the available memory.

Next, we analyze the CPU cost of our algorithm. Let us assume that each pair of items will have an equal probability to satisfy the join predicate and that the number of input items will be N. Then, it follows that the total number of answers (including the ones that are produced multiple times) is at most

$$Sel_{join}(R,S) * N^2 * F/(F-1).$$

Here F denotes the fan-in of the merge and Sel_{join} denotes the selectivity of the join. This is simply because the number of answers will increase by a factor F for each level of the merge.

In our current approach to eliminating duplicates we run for each element that has been touched during a merge $F-1$ queries against a sweep area. With the same arguments as above, it follows that at most

$$N * F/(F-1)$$

items are touched during merge. Therefore, PMJ requires $N * F$ queries against sweep areas whereas the traditional sort-merge join requires only N queries. The runtime can be alleviated by running queries only against the non-empty sweep areas.

3.5 Quality of Samples

In this section we present an estimator for computing the selectivity of the joins. Moreover, we show the accuracy of the estimator by providing confidence intervals. The following approach is based on the ideas of survey sampling [2], which is different to the ones previously published in the database area [12]. Survey sampling corresponds to sampling without replacement.

Without loss of generality, we assume that the merge tree consists of two levels only (root and leaf level). Due to the query processing of PMJ, $R \times S$ is distributed into blocks of size $(M/2)^2$, i.e., there are at most $M^2/4$ answers to the join per block. A block corresponds to the data that is processed when PMJ is called for two initial runs. We assume in the following that R and S are in random order and that the order of R is independent from the order of S. Let us consider a tuple $(r,s) \in R \times S$. The probability that (r,s) occurs in one of the blocks is the same for each block. This is a direct consequence from the independence of R and S. It follows that the data from an arbitrary sequence of k blocks represents a random sample of size $k \cdot M^2/4$ of $R \times S$. The basic idea of our approach is to count the qualifying tuples in a block by applying the following indicator function:

$$I(r,s) = \begin{cases} 1 & \text{if } P_{join}(r,s) \\ 0 & \text{otherwise} \end{cases} .$$

This refers to an experiment without replacement (Laplace experiment) and can be modeled by the hyper-geometric distribution. Let C be a set of k blocks and $n = k \cdot M^2/4$ be the current size of the sample within the blocks. We propose to use the following estimator for the mean:

$$\hat{y} = \frac{1}{n} \sum_{b \in C} \sum_{(r,s) \in b} I(r,s). \tag{3}$$

It follows from the hyper-geometric distribution that the estimator converges to the mean; the estimator is then called *unbiased*. In the following, we calculate for a given α the corresponding confidence interval of the estimator with respect to the Central Limit Theorem. The confidence interval is a range where the actual mean of the entire response set will be with probability $1 - \alpha$ (for a sufficiently large sample). This requires the computation of the variance of $\hat{y}$.

THEOREM 3 (VARIANCE OF ESTIMATOR).
Let $T = |R \times S|$. Let n be the size of a sample independently and randomly drawn from $R \times S$ without replacement. For a join predicate P_{join}, the variance of the estimator is then given by

$$Var(\hat{y}) = \left(1 - \frac{n}{T}\right) V^2/n$$

where V denotes the (deterministic) variance of $I(r,s)$, $(r,s) \in R \times S$.

PROOF (VARIANCE OF ESTIMATOR). See [2]. $\square$

Since V is not known in advance, we use the following unbiased estimator instead:

$$\hat{V}^2 = \frac{1}{n-1} \sum_{b \in C} \sum_{(r,s) \in b} \left(I(r,s) - \hat{y}\right)^2 .$$

Overall, this results in the following unbiased estimator of the variance:

$$\widehat{\mathrm{Var}}(\hat{y}) = \left(1 - \frac{n}{T}\right) \hat{V}^2/n.$$

In an asymptotic sense we are able to apply the Central Limit Theorem and to calculate the following $(1-\alpha)$-confidence interval for $\hat{y}$:

$$\left(\hat{y} - z_\alpha \hat{V} \sqrt{\left(1 - \frac{n}{T}\right)/n},\ \hat{y} + z_\alpha \hat{V} \sqrt{\left(1 - \frac{n}{T}\right)/n}\right) \tag{4}$$

where z_α refers to the α-quantile of the standard normal distribution $N(0,1)$.

Due to the sorting of two initial runs before they are joined, the ordering of the data within a block is not random for PMJ. At the end of the processing of an entire block, however, the sequence of data refers to a random sample again. Therefore, we are then and only then in the position to calculate a confidence interval. In order to increase the update frequency of the confidence interval, it is possible to change the in-memory processing of PMJ. PMJ can be applied in memory to smaller blocks which are not written to disk but merged immediately. More details on the processing of the confidence intervals are given in the full version of the paper [9].

Let us compare our approach to the one proposed for the ripple join [11, 12]. The advantage of our estimator and its associated confidence interval is their low overhead and inexpensive computation (about two arithmetic operations for each answer of the join). For the ripple join, the computation of the confidence interval is more expensive with respect to both time and space. In particular, the worst-case occurs for the ripple join when the frequency of the join values is very low. This is typically satisfied for spatial and similarity joins. The overall cost of a ripple join is $O(n^2)$, whereas the cost of PMJ is $O(n \log n + k)$ for many types of joins (e.g. equi-join) where k denotes the number of answers. Moreover, our selectivity estimator is generic in the sense that it can easily be applied to all kinds of join predicates where the evaluation of the predicate only depends on the two input items. In particular, similarity joins and spatial joins are fully supported at the same cost as other joins. To the best of the authors knowledge, we are not aware of other sampling-based (non-parametric) methods applied to estimating the selectivity of similarity joins and spatial joins where confidence intervals control accuracy. A disadvantage of our approach compared to the ripple join seems to be the lower update frequency of the confidence interval. This however will be outweighed by the fact that our estimator is more inexpensive to compute. Moreover, as mentioned above, the update frequency can be improved by changing the in-memory processing of PMJ or using a smaller fraction of the the available memory.

3.6 Important considerations

There are different kinds of possible extensions of PMJ that are important to mention and worth for being examined in our future research.

First, PMJ is not limited to the computation of an inner join, but can be extended to compute different kinds of outer joins and semi joins. For semi joins of relations R and S, we gain a performance advantage in comparison to the original approach. Whenever PMJ generates a result tuple $r \in R$, there is no need anymore to append r to the next run (see for example line 12 of Figure 3). This reduces the I/O cost significantly in case of correlated input sets.

Second, the memory occupation of PMJ is not fixed, but the amount of occupied memory can be adjusted dynamically during runtime. This is advantageous when the available memory space varies. The techniques known from memory-adaptive sorting [31] are also applicable to PMJ. We can even partition memory unevenly among input relations so as to get early join samples with highest significance, as required in ripple joins [11].

Third, the production of the first early results may suffer from the fact of large main memories. Large memories would also result in unacceptable update frequencies of the online selectivity estimator and its confidence interval. There are two solutions to reducing the time until results are produced (again). First, it is possible to set M smaller than the available main memory would allow. However, this reduces the overall efficiency of the join. An interesting strategy is therefore to start PMJ with a small value of M and increase M up to its maximum value during the early processing. A different approach is to use a combination of mergesort and quicksort for sorting in memory. First, quicksort is applied to small subsets (which can be sorted within a second) and then mergesort is used to produce sorted subsets of size M. This would give us the advantage of reporting join results after quicksort has produced a pair of sorted subsets. This basically resembles the external strategy of PMJ.

Fourth, when the output should be delivered in sorted order with respect to the join attribute, it still might be worthwhile to run a slightly modified version of PMJ up to the point where the last merge has to be performed. This actually corresponds to the open phase of the join operator. As it is known from ordinary sort-merge joins, results are only produced during the last merge and therefore, they are delivered in sorted order. Note that the overall I/O cost of this approach is only slightly higher, assuming that the sweep area always fit into memory. The advantage of this approach is that a selectivity estimator (and/or other statistics) can be computed from the early answers that are not reported to the user. The estimated value can then be output before the first regular result will be sent. Consider for example that we inform the parent operator in the operator tree about the expected number of tuples. This value will almost always be more accurate than the first estimation of the optimizer. Therefore, performance-critical parameters of the parent operator can still be changed based on the more accurate estimation.

Fifth, PMJ is also applicable to processing joins on more than two input relations. An obvious and generally applicable approach is to build up a pipeline of binary joins (and to distribute memory among the joins carefully). A more advanced technique can be applied when join predicates refer to the same set of attributes. The basic idea is then to sort all input relations simultaneously and to produce results as early as the first initial runs are sorted. This problem will be addressed in our future research.

In general, PMJ can be improved further by applying the broad range of techniques known from external sorting. This is true in particular for parallel processing. For example, the technique of balancing the load among processors by efficiently computing quantiles and then distributing sequences accordingly [15] carries over with small modifications. This does not take care of balancing the load induced by join result tuples — an interesting open problem.

4. EXPERIMENTS

In this section we present experimental results for three types of joins (equi join, spatial join, similarity join) by using different sort-based algorithms. Among these algorithms are PMJ, the strict sort-merge-join (*strict SMJ*) [4] and the semi-strict sort-merge-join (*semi-strict SMJ*) [20]. The purpose of the experiments is to demonstrate the broad applicability of PMJ for various types of joins. Moreover, we also want to report the peroformance of PMJ in comparison to its blocking counterparts.

All algorithms were implemented in Java 1.4 on top of XXL [3], a freely available query processing library. The implementations are generic in the sense that the different join algorithms are supported within a single implementation. This is accomplished by using different classes of sweep areas. Moreover, Replacement Selection is our default method for sorting in-memory, due to its advantage of producing longer initial runs than Quicksort. The join algorithms also satisfy the Iterator interface, where a call of the method `next` delivers the next result tuple of the join.

All experiments run on the Java HotSpot Server Virtual Machine and a 700 MHz Athlon processor with 256MB of memory. We measured the number of disk accesses as well as the total runtime. We charge one I/O for each write or read of a page. The page size in the experiments was set to 4 KB. We assume that the algorithms are part of an operator tree. Therefore, the cost for the initial read of the data sets as well as the cost for delivering result tuples to the next operator – which are the same for all algorithms – are not considered. In order to measure the total runtime, the buffer of the operating system was turned off by using raw devices. The runs that are generated during sorting are stored contiguously on disk. As a default value, we set the available main memory to 10% of the size of the input sets. Therefore, our experiments refer to the case where the merge-tree consists of only two levels. Due to the availability of large main memories, merge trees with height greater than two will seldom occur in practice.

In our experiments, we used real-world data sets as well as synthetically generated ones. In Table 2 we have listed for each join type the data sets we have used in the experiments. For the equi-join, we used two uniformly distributed data sets, whereas for the other join types we used the data sets (ST, RR) obtained from the TIGER files [21] and data sets from a CAD application (CAD1, CAD2, CAD1_Shuf, CAD2_Shuf).

4.1 Comparison of Join Algorithms

In this section we report the results obtained from an experimental comparison of the different algorithms (PMJ, strict SMJ, semi-strict SMJ). We report the number of result tuples computed by the join operators as a function of time and number of I/Os.

4.1.1 *Equi-Join*

In our first experiment we computed the equi-join between UNI1 and UNI2 (Table 2). Each set consists of 2, 000, 000 uniformly distributed integers. The results are plotted in Figure 6 where on the left hand side and right hand side the number of results tuples are reported as a function of I/Os and runtime, respectively. Strict SMJ and semi-strict SMJ start producing results after more than 11000 I/Os and 4000 I/Os, respectively, whereas PMJ returns results from the very beginning. It is interesting that the curve of PMJ is a piece-wise linear function. The first linear piece refers to the run generation phase, whereas the other refers to the merge phase.

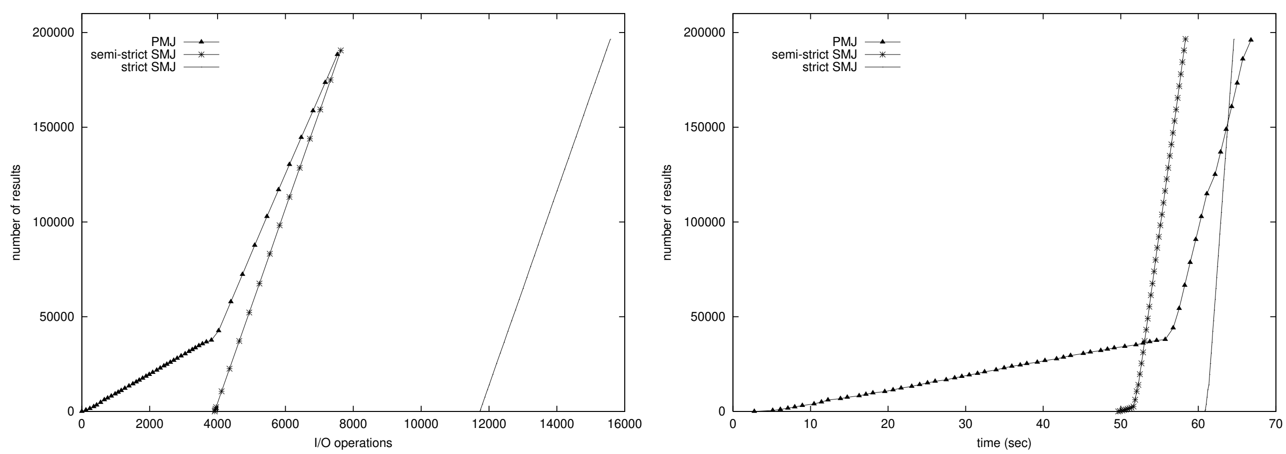
The results that depend on the total runtime (see Figure 6b) are similar for strict SMJ and semi-strict SMJ. Both algorithms start producing results only after 50 (61) seconds. Thereafter, both algorithms produce the results very fast. In contrast, PMJ delivers the first result tuple in less than a second. It reports a linearly increasing number of results until the run generation phase is finished. After that the curve is again linear but steeper than the one related to the run generation phase. This shows the faster speed at which answers are produced.

4.1.2 *Spatial Join*

In the following, we discuss the results of experiments where the spatial join is performed on data sets ST and RR. The sweep area is based on *hashing* as it is described

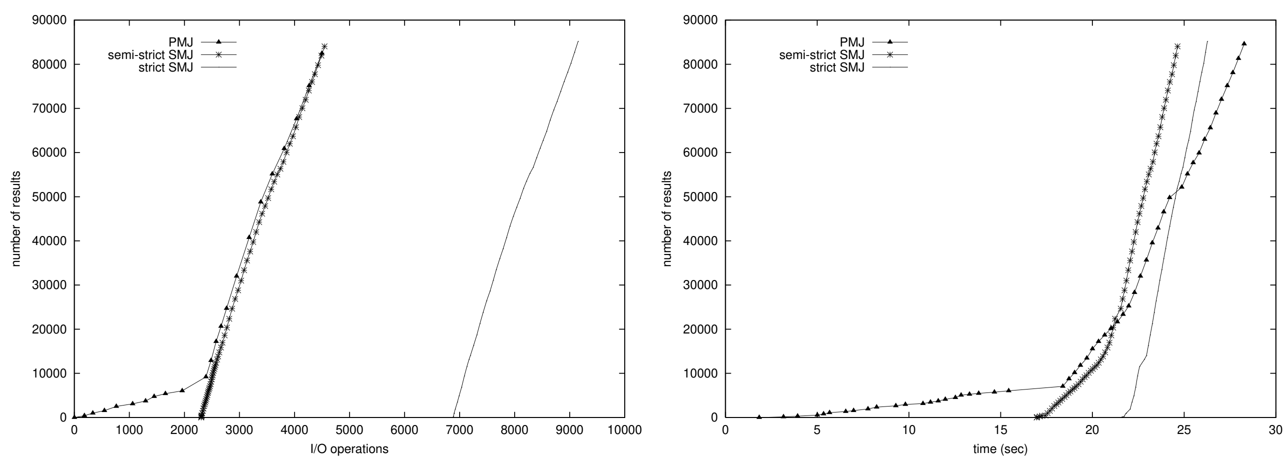
join	data set	description	data type	#items	dim	size [MB]
equi-join	UNI1	uniformly distributed	integers	2,000,000	1	8.0
	UNI2	uniformly distributed	integers	2,000,000	1	8.0
spatial join	ST	street segments of L.A.	rectangle	131,461	2	4.5
	RR	rail&road segments of L.A.	rectangle	128,971	2	4.4
similarity join	CAD1	fourier-vectors of CAD-parts	point	657,048	16	40.1
	CAD2	fourier-vectors of CAD-parts	point	655,125	16	40.0
	CAD1_Shuf	CAD1 shuffled	point	657,048	16	40.1
	CAD2_Shuf	CAD2 shuffled	point	655,125	16	40.0

Table 2: The data sets used in the experiments



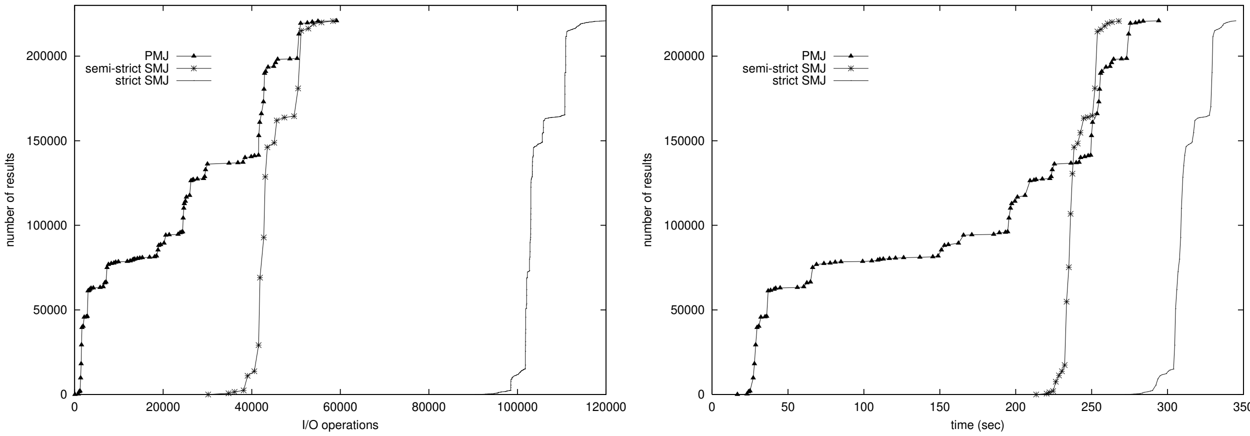
(a) I/O (b) time

Figure 6: Equi-join UNI1 ⋈ UNI2: Number of result tuples as a function of I/O and time



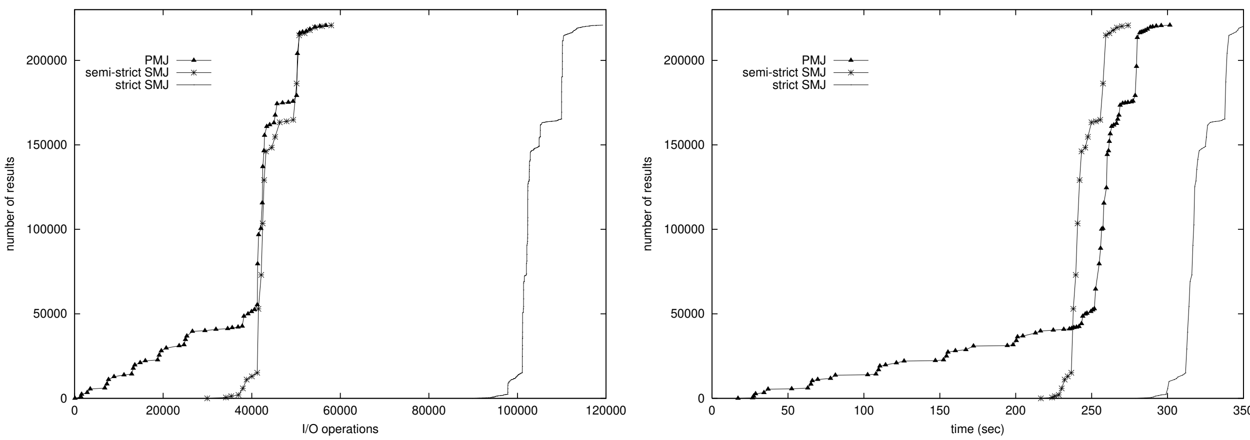
(a) I/O (b) time

Figure 7: Spatial join ST ⋈ RR: Number of results as a function of I/O and time



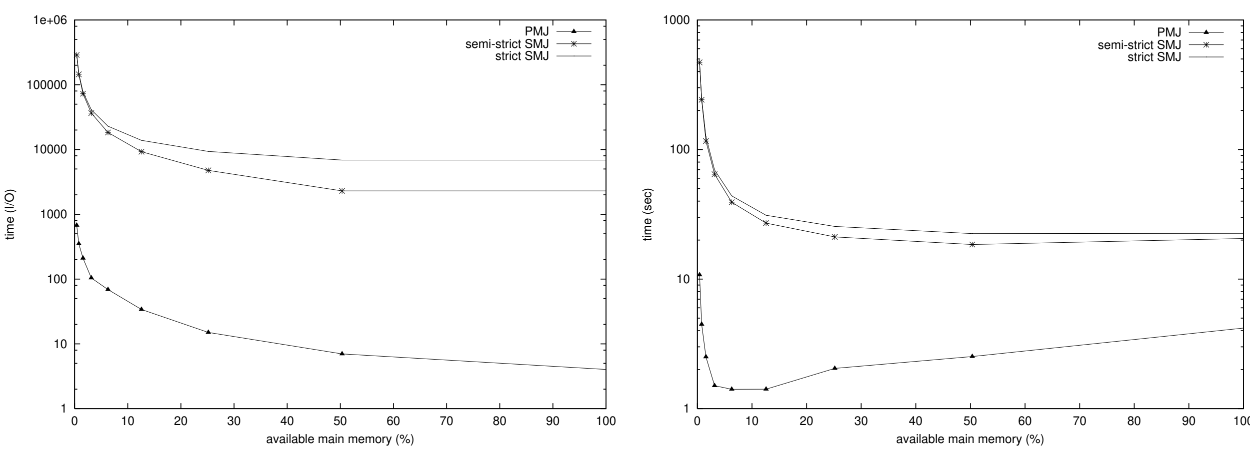
(a) I/O (b) time

Figure 8: Similarity join CAD1 $\bowtie$ CAD2



(a) I/O (b) time

Figure 9: Similarity join CAD1_Shuf $\bowtie$ CAD2_Shuf



(a) I/O (b) time

Figure 10: Time needed to report the first 100 answers of the spatial join ST $\bowtie$ RR

in [1]. Despite the fact that strict SMJ was used in the experiments of [1], the method has already been shown to be more efficient than its competitors like PBSM [24].

The graphs in Figure 7 show that PMJ again reports a considerable number of result tuples from the very beginning. When semi-strict SMJ starts reporting results, PMJ has already produced more than 8,000 answers. The total number of I/Os is almost the same for PMJ and semi-strict SMJ, whereas strict SMJ requires double the number of I/Os. In Figure 7b the number of results is reported as a function of the runtime. PMJ reports tuples from the beginning whereas the two competitors start their production after 17 and 22 seconds, respectively. The speed of result production is however higher for these methods compared to PMJ. This can be explained by the additional overhead for duplicate avoidance that requires issuing queries on different sweep areas. The overhead of the total runtime is only one third of the runtime of semi-strict SMJ.

4.1.3 Similarity Join

Figure 8 shows the result of an experiment where a similarity join is computed on the 16-dimensional data sets CAD1 and CAD2. These data sets are generated from splitting a data set into two. This is performed such that a tuple is assigned with probability 0.5 to CAD1, otherwise to CAD2. Due to clustering effects in the original data set, there is a strong correlation between CAD1 and CAD2. This is the reason why the total runtime of PMJ is only slightly higher than the one of SMJ and that many results are already produced during the initial run generation (Figure 8). After 70 seconds one third of the answers have been already reported, whereas semi-strict SMJ reports the first answer only after 215 seconds. PMJ has no overhead in terms of I/O for this experiment.

In a second experiment we computed the similarity join for shuffled versions of CAD1 and CAD2. The results are plotted in Figure 9. The results for strict and semi-strict SMJ look similar to the ones in the previous experiment. The results for PMJ however are different. Only a modest number of results are produced during run generation. In comparison to the experiments with equi-joins and spatial joins, we observe a much higher production rate of the results during the merge phase. Thus, the total runtime of PMJ is almost the same as in the previous experiment.

4.2 The impact of the memory size

In this experiment we varied the main memory available to the algorithms. We report the time required to compute the first 100 tuples of a spatial join between data sets RR and ST as a function of the available memory (Figure 10). The size of the memory is given relative to the size of the data sets. For all buffer sizes, PMJ reports the first 100 tuples by at least one order of magnitude earlier than semi-strict SMJ. For small buffers ($< 5\%$) this improvement rises up to two orders of magnitude and higher.

4.3 Selectivity Estimation

In this section, we present results that illustrate the quality of our online selectivity estimator (see Section 3.5). For this set of experiments, Quicksort is used for in-memory sorting to avoid data skew in the initial runs. In Figure 11 we report the results of an experiment where a similarity join is computed for data sets CAD1_Shuf and CAD2_Shuf.

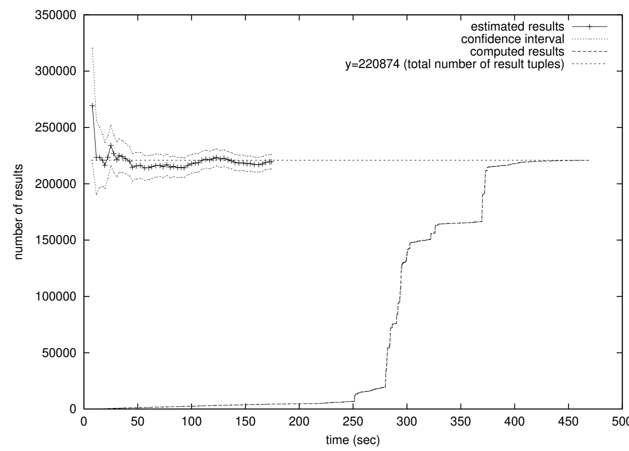


Figure 11: Selectivity estimation for the similarity join CAD1_Shuf $\bowtie$ CAD2_Shuf (memory size 2%)

In order to compute frequent updates of the estimation, we set the available memory to 2% of the size of input sets. Note that this causes a higher overall runtime compared to the experiment of Figure 9 where the available memory was greater by a factor of 5. Figure 11 depicts the estimated value and its 95% confidence interval as a function of the total runtime. After a few seconds, the estimated value is already very close to the total number of answers. The accuracy of the estimation is well documented by the confidence interval. The fourth curve in the figure depicts the number of results (as a function of the runtime).

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have addressed the problem of designing efficient non-blocking algorithms for join processing that support an early and progressive production of results. Our primary target was to support the production of early results for advanced joins like spatial and similarity joins. Since all state-of-the-art algorithms are based on the sort-merge paradigm, we put our focus on this kind of methods.

We proposed a new non-blocking sort-based join technique called Progressive Merge Join (PMJ). PMJ is broadly applicable to processing a join based on the sort-merge paradigm whenever a blocking counterpart exists. The basic idea of PMJ is to produce first results of the join in parallel to sorting the input sets. Sorting is done simultaneously on both input sets. We identified the conditions that have to be satisfied by a specific type of join, in order to run under our generic framework that employs a so-called sweep area for in-memory processing. Moreover, it is formally proved that the generic framework of PMJ is correct. We showed that different types of joins can be processed by specialized variations of PMJ. Among those are equi-joins, temporal joins, band joins, spatial joins and similarity joins. All these variants are non-blocking methods with the emphasis on early production of results. To the best of the authors' knowledge, PMJ is the first approach to computing early results for the similarity join, a join type widely used for data mining. Moreover, we also showed that the first results are sufficiently representative for computing an online selectivity and its associated confidence interval. Therefore, PMJ is particularly useful for the interactive exploration of large data sets where users may abort join processing when the selectivity of the join does not match their expectation. We also examined the total runtime of PMJ which is shown to be almost the same as the runtime of its blocking counterpart. Finally, we compared the results obtained from experiments on large sets of different data. Our results confirm that PMJ produces results much earlier than its blocking counterparts, while the total runtime of PMJ is only slightly higher.

In terms of future work, we plan to examine the problem for multiple input relations and to develop an analytical framework that provides answers to fundamental question on how fast first join results can be produced. There are also many possible extensions to PMJ with respect to its implementation. All the techniques can be applied to PMJ that are originally designed for improving the runtime of external sorting.

Acknowledgements

We would like to thank Björn Blohsfeld and Alexander Markowetz for the helpful suggestions, and Christoph Freytag and the anonymous reviewers for the insightful comments.

6. REFERENCES

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *VLDB*, pages 570–581, 1998.

[2] V. Barnett. *Sample Survey — Principles & Methods*. Edward Arnold, 1991.

[3] J. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL — A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *VLDB*, pages 39–48, 2001.

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

[5] C. Böhm, B. Braunmüller, M. M. Breunig, and H.-P. Kriegel. High Performance Clustering Based on the Similarity Join. In *CIKM*, pages 298–313, 2000.

[6] 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*, pages 379–388, 2001.

[7] J.-P. Dittrich and B. Seeger. Data Redundancy and Duplicate Detection in Spatial Join Processing. In *ICDE*, pages 535–546, 2000.

[8] J.-P. Dittrich and B. Seeger. GESS: a Scalable Similarity-Join Algorithm for Mining Large Data Sets in High Dimensional Spaces. In *ACM SIGKDD*, pages 47–56, 2001.

[9] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer. Progressive Merge Join (full version). in preparation, 2002.

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

[11] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *ACM SIGMOD*, pages 287–298, 1999.

[12] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and Cost Estimation for Joins Based on Random Sampling. In *JCSS*, volume 52, pages 550–569, 1996.

[13] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In *ACM SIGMOD*, pages 171–182, 1997.

[14] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. *ACM SIGMOD*, pages 299–310, 1999.

[15] B. R. Iyer, G. R. Ricard, and P. J. Varman. Percentile finding algorithm for multiple sorted runs. In *VLDB*, pages 135–144, 1989.

[16] N. Koudas and K. Sevcik. Size Separation Spatial Join. In *ACM SIGMOD*, pages 324–335, 1997.

[17] N. Koudas and K. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *TKDE*, 12:3–18, 2000.

[18] H. Lu and K.-L. Tan. On Sort-Merge Algorithm for Band Joins. *TKDE*, 7(3):508–510, 1995.

[19] G. Luo, J. Naughton, and C. Ellmann. A Non-blocking Parallel Spatial Join Algorithm. *ICDE*, page to appear, 2002.

[20] M. Negri and G. Pelagatti. Join during merge: An improved sort based algorithm. *IPL*, 21(1):11–16, 1985.

[21] B. of Census. TIGER/Line precensus files (1996 Technical Documentation), 1996.

[22] J. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *ACM SIGMOD*, pages 326–336. ACM Press, 1986.

[23] J. Orenstein. An Algorithm for Computing the Overlay of k–Dimensional Spaces. In *SSD*, pages 381–400, 1991.

[24] J. Patel and D. DeWitt. Partition Based Spatial-Merge Join. In *ACM SIGMOD*, pages 259–270, 1996.

[25] F. P. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[26] L. Raschid and S. Y. W. Su. A parallel processing strategy for evaluating recursive queries. *VLDB*, pages 412–419, 1986.

[27] K. Shim, R. Srikant, and R. Agrawal. High-Dimensional Similarity Joins. In *ICDE*, pages 301–313, 1997.

[28] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. *ICDE*, pages 282–292, 1994.

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

[30] A. Wilschut and P. Apers. Pipelining in Query Execution. *Conference on Databases, Parallel Architectures and their Applications, Miami, USA*, pages 68–77, 1991.

[31] W. Zhang and P.-Å. Larson. Dynamic memory adjustment for external mergesort. In *VLDB*, pages 376–385, 1997.