Plug&Join: An easy-to-use generic algorithm for efficiently processing equi and non-equi joins

Jochen van den Bercken, Martin Schneider and Bernhard Seeger

Department of Mathematics and Computer Science
University of Marburg
Hans-Meerwein-Straße, 35032 Marburg, Germany
{bercken,maschn,seeger}@mathematik.uni-marburg.de

Abstract

This paper presents Plug&Join, a new generic algorithm for efficiently processing a broad class of different types of joins in an extensible database system. Plug&Join is not only designed to support equi joins, temporal joins, spatial joins, subset joins and other types of joins, but in contrast to previous algorithms it can be easily customized and it allows efficient processing of new types of joins that might be of relevance in the near future. Depending on the join predicate (and the data types of the join relations) Plug&Join is called with a suitable type of index structure as a parameter. Fortunately, custom types of index structures can be implemented easily under frameworks like GiST which simplifies and extends the applicability of our approach.

Plug&Join partitions both join relations recursively until each partition of the inner relation fits in main memory. If an inner partition fits in memory, the algorithm builds a memory resident index of the desired type on the inner partition and probes all tupels of the corresponding outer partition against the index. Otherwise, a memory resident index is created by sampling the inner partition. The index is then used as a partitioning function for both partitions.

In order to demonstrate the flexibility of Plug&Join, we present how to implement equi joins, spatial joins and subset joins by using memory resident B+-trees, R-trees and S-trees, respectively. Moreover, results obtained from different experiments for the spatial join show that Plug&Join is competitive to special-purpose methods like the Partition Based Spatial-Merge Join algorithm (PBSM).

1 Introduction

In order to cope with large sets of complex data, database management systems (DBMS) have to be equipped with new methods for processing queries. While query processing techniques for different application areas like spatial databases have already been integrated into commercial DBMS, there are also serious concerns about such special-purpose solutions. In particular, the complexity of current DBMS

increases dramatically leading to all the well-known problems of software development and maintenance. While DBMS provide some advanced query processing techniques, most of the systems are still too inflexible since they are not intended for receiving new functionality from the user (or if so, this is too difficult). Although the idea of such an extensible system is well known in the database community since almost two decades, we still have the impression that there is a lack of basic generic query processing techniques.

In this paper, we reconsider the classical problem of join processing: For relations *R* and *S* and a predicate *p* the result set of a join *R p S* consists of those tuples from the cartesian product of *R* and *S* which satisfy *p*. We are interested in *generic* and *efficient* methods for join processing which can be used in different application areas such as spatial databases (spatial join), temporal databases (temporal join), object-oriented databases (subset join) and others. Although the nested-loops join is an example of a generic method, it clearly fails to be efficient. Unfortunately, the methods proposed for equi joins (and which are implemented in todays commercial DBMS) cannot be used for supporting other types of joins.

In the following, we present a new generic join algorithm called Plug&Join which is applicable to a large number of joins including the ones mentioned above. Plug&Join is a generic algorithm that is parameterized by the type of an *index structure*. Note however that Plug&Join does *not* require a preexisting index on any of the relations. Depending on the specific index structure, Plug&Join is able to support different types of join predicates. We also show that it performs really fast. For spatial joins, we compare the performance of Plug&Join with the Partition Based Spatial-Merge Join (PBSM) [PD 96] which is considered to be among the most efficient join methods to compute spatial joins. Plug&Join using R*-trees [BKSS 90] is shown to be competitive to an improved version of PBSM [DS 99] which runs considerably faster than the original version.

The remainder of this paper is organized as follows. First, we will give a review of previous work on join processing where our focus is put on generic algorithms. Section 2 introduces our model of the underlying index structure. Thereafter, we present our generic algorithm Plug&Join in more detail. In Section 3, we discuss several use-cases for Plug&Join. We start by explaining how to support equi joins and present

thereafter the use-cases for spatial joins and subset joins. Section 4 presents an experimental comparison of Plug&Join and PBSM using non-artificial spatial data sets.

1.1 Review of previous work

Good surveys of algorithms for processing relational joins are provided in [Sha 86], [ME 92] and [Gra 93]. There has been quite a lot of work on spatial joins recently ([Ore 86], [BFH 93], [BKS 93], [Gün 93], [LR 94], [HS 95], [LR 96], [PD 96], [KS 97], [APR+ 98], [MP 99]). The spatial join combines two sets of spatial objects with respect to a spatial predicate. Most of the mentioned work has dealt with intersection as the join predicate, but there is also the need to support other predicates [PTSE 95]. In temporal databases, so-called temporal joins are required ([SSJ 94], [Zur 97]). A temporal join is basically a one-dimensional spatial join, but because of the high overlap of the data in temporal databases, most of the methods known from spatial joins perform poorly. In object-oriented databases, we need to support subset joins [HM 97]. The join predicate of the subset join is defined on two set-valued attributes, say R.A and S.B, and all tuples of the cartesian product are retrieved where $R.A \supseteq S.B$ holds. Most of these join algorithms have in common that they are designed to support one specific join operation and it is not evident how to extend their functionality to support other joins.

Generic frameworks for query processing techniques have attracted only little attention in the literature, so far. The GiST approach [HNP 95] consists of a framework for dynamic tree-based index structures which supports user-defined search predicates. Index structures like B+-trees, R-trees [Gut 84] and M-trees [CPZ 97] can be derived from GiST. Actually, GiST is an ideal candidate for initializing Plug&Join as we'll explain in section 2.1. The Bulk Index Join [BSW 99] is a framework for join processing, but it assumes that a preexisting index has been created for one of the relations. In contrast, Plug&Join does not require a preexisting index for any of the relations.

Most related to Plug&Join is the Spatial Hash Join (SHJ) [LR 96] of Lo and Ravishankar. The SHJ is primarily designed to support spatial joins, but the framework can easily be extended to support join predi-

cates with a spatial relationship in a one- or multidimensional data space. The SHJ is a divide&conquer approach that partitions both of the relations into buckets. Lo and Ravishankar provide a classification according to the criteria whether the objects of the two input relations are assigned to one or multiple buckets. Plug&Join differs from SHJ for the following reasons: First, Plug&Join is more general since it can also be applied to join predicates like the subset predicate where the predicate is not related to a spatial domain. Second, our approach is more specialized since we do not allow data of one of the input relations to be assigned to multiple buckets. The main reason for this restriction is that otherwise duplicates may arise in the result set which lead to a substantial performance reduction when the join selectivity is high. Third, and most important, our generic algorithm is able to use index structures available in the DBMS to partition the data, whereas in SHJ the partitioning function still has to be implemented. We believe that the implementation is however too difficult and therefore, SHJ is not an adequate generic approach to an extensible DBMS.

2 Plug&Join

In this section we give a full description of Plug&Join. First, we discuss why we use index structures as a basis for our generic join algorithm. Thereafter, we give a detailed description of Plug&Join.

2.1 A plea for index structures

Plug&Join partitions both join relations recursively until each partition of the inner relation fits in main memory. Therefore, the algorithm needs a partitioning function which has to be provided as a parameter. If an inner partition fits in memory, the algorithm has to join the inner partition with its corresponding outer partition. Thus, an appropriate main memory join algorithm has to be available.

We are in the position to cope with both requirements by using index structures: First, a partitioning function can simply be created by inserting a sample of the inner relation into a memory resident index of the desired type. Each leaf of the index then corresponds to a pair of partitions of the join relations. Second, a

simple but efficient main memory join algorithm can be obtained in a similar way: We build a memory resident index of the desired type on the inner partition and probe all tupels of the corresponding outer partition against that index.

In the following, we assume an index structure to be a *grow-and-post tree* [Lom 91]. We will give the reasons shortly after the description of grow-and-post trees.

Grow-and-post trees contains of two types of nodes, see Figure 1: internal nodes and leaf nodes. Each of the

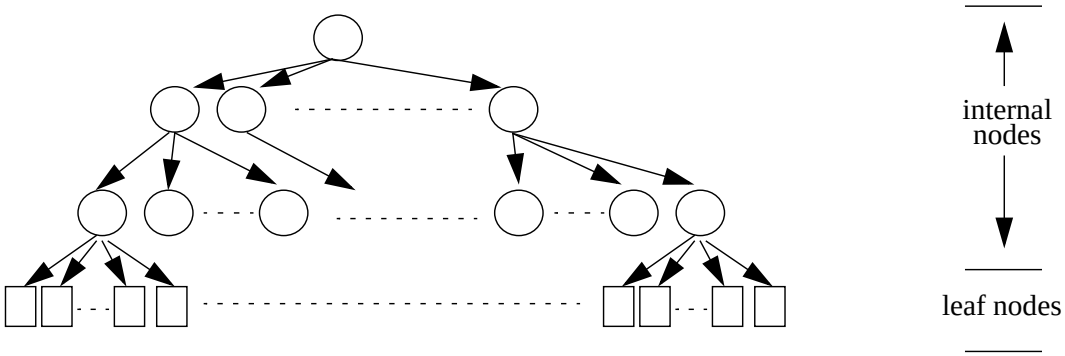

Figure 1: Structure of a grow-and-post tree

internal nodes consists of a routing table with at most *C* entries referencing a subtree. A leaf node refers to a page on disk, i. e., there are at most *B* tupels stored in a leaf node.

As known for B+-trees, an insertion of a new tupel into a grow-and-post tree only requires the nodes from a single path of the tree. First, the tupel traverses the tree from the root to a leaf node in which the tupel is inserted into, i. e., the node *grows* in size. In case that a node overflows, it is split into two nodes, and a new entry has to be inserted into the parent node. This may again result in an overflow of the parent node, which is treated analogously. In terms of grow-and-post trees, index entries are *posted* in direction up to the root.

There are three reasons for choosing grow-and-post trees as the abstract index structure for Plug&Join:

First, in addition to B+-trees, many advanced index structures like the R-tree [Gut 84], S-tree [Dep 86], LSD-tree [HSW 89], M-tree [CPZ 97] and the multiversion-B-tree [BGO+ 96] are grow-and-post trees. Some of them like B+-trees or R-trees are already implemented in DBMS or geographical information systems. These preimplemented index structures generally support a large set of different query types and therefore can also be used in combination with Plug&Join to support different types of joins. This does not

hold in general for other kind of index structures like hashtables which are primarily designed to support exact match queries. Hashtables are therefore restricted to only support equi joins.

Second, frameworks like GiST allow to implement custom types of grow-and-post trees with little effort. Furthermore, it is even simpler in GiST to provide a new type of query for an index structure that has been implemented under GiST [Aok 98].

Third, grow-and-post trees are data-adaptive because of their ability to split full nodes. Therefore, an index created using a sample of a relation allows, with high probability, Plug&Join to partition the relation evenly. Note that grow-and-post trees have a kind of build-in adaption to data distributions, i. e. no specialized sampling technique has to be provided. Consequently, Plug&Join using grow-and-post trees will generally alleviate the problem of data skew and reduce the overhead of repartitioning compared to using hashtables or other static index structures.

2.2 The generic algorithm

In this section, we give a detailed description of our algorithm Plug&Join. The algorithm depends on five input parameters. The first two parameters relate to R and S, the input relations of the join. As a third parameter, the user has to provide an appropriate type of index structure which meets our requirements of a grow-and-post tree. Note that Plug&Join is not restricted to grow-and-post trees, but as described in section 2.1, this class of index structures has several advantages compared to other index structures like hashtables. The maximum fan-out of the internal nodes of the index structure and the capacity of the leaves are specified by the last two parameters.

The algorithm uses indices of the desired type to partition the data of both relations where one pair of partitions corresponds to one leaf. In order to determine the maximum numbers of partitions (leaves), the function *computeMaximumLeavesNumber* is called once for each invocation of Plug&Join. This function requires as input parameters the inner relation *R*, the type of the index structure *TreeType*, its fan-out and the capacity of its leaves. For a leaf *l*, we use the terminology *l.Lpartition* and *l.Rpartition* to refer to the

corresponding partitions that contain tuples from R and S, respectively.

Algorithm **Plug&Join** (*R, S, TreeType, fanOut, leafCapacity*)

```
if (S is not empty)
    maxLeavesNumber := computeMaxLeavesNumber(R, TreeType, fanOut, leafCapacity)
    index := createTree(TreeType, fanOut, leafCapacity)
    while R is not empty
        remove one tupel r from R
        if number of leaves of index <= maxLeavesNumber
            insert r into the index
        else
            find the leaf of the index into which r would have to be inserted
            if leaf overflows
                insert r into leaf.Rpartition
            else
                insert r into leaf
    foreach leaf of index
        if leaf.Rpartition is not empty
            remove all tupels from leaf and insert them into leaf.Rpartition
            flush leaf.Rpartition to disk
    while S is not empty
        remove one tupel s from S and transform it into a query
        foreach leaf of index that is examined by the query
            if leaf is empty
                insert s into leaf.Spartition
            else
                report all tupels in leaf that answer the query
    foreach leaf of index
        if leaf.Spartition is not empty
            flush leaf.Spartition to disk
    delete the index
    foreach pair (Rpartition, Spartition) produced before
        Plug&Join (Rpartition, Spartition, TreeType, fanOut, leafCapacity)
```

The algorithm works as follows:

Plug&Join first creates a new index of the desired *TreeType*. Then, it reads from the inner relation R the tuples one by one and inserts them into the index. In order to support this step efficiently, the index is kept resident in main memory and therefore, no I/Os are performed.

First, let us discuss the case when all of the records of relation R fit in main memory. We therefore do not have to limit the number of leaves of the index to be created, thus the function *computeMaximum-LeavesNumber* will return ∞. After having built up the index in main memory, the tupels of the outer relation S are transformed into queries which are processed by the index immediately. The set of results obtained from all queries corresponds to the total result set of the join.

Second, let us consider the case when R does not fit in main memory. Then, we have to partition both relations R and S pairwise, see Figure 2. Each pair of partitions corresponds to one leaf of the index, so the

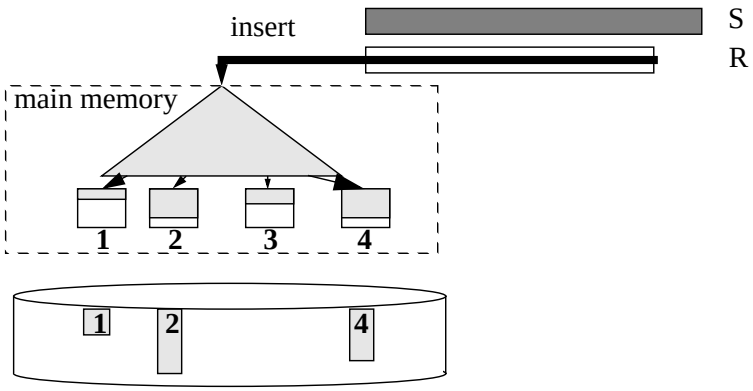

Figure 2: Processing the inner relation

function *computeMaxLeavesNumber* determines how many pairs of partitions to create. On one hand, the number of pairs has to be big enough to minimize repartitioning. On the other hand, choosing the number too big increases random I/O, thus the performance degrades. As a first approach, we adopt the rule of PBSM, see section 4.1. Due to space limitations, we'll examine the problem in our future work. As long as the number of leaves of the index has not reached the threshold m*axLeavesNumber*, the tupels of R are just inserted into the index. Recall that, by assumption, the index stores tupels in exactly one of its leaves, thus the tupels of relation R will not be replicated. As soon as the number of leaves of the index reaches the pre-

computed threshold, an overflow in one of the leaves does not result in a structural change anymore, but the tupel (which causes the overflow) is simply written into the associated Rpartition. Each partition is implemented as a simple queue by a linked list of buckets on disk. The access to a queue is buffered in such a way that its last bucket is kept in main memory. In case that this bucket is also full, it is flushed to disk. The size of a bucket is given by the available main memory divided by the number of leaves.

After all tupels of R have been processed, we examine each leaf of the index. In case that its corresponding Rpartition is not empty, we remove all tupels from that leaf, insert them into its Rpartition and flush the Rpartition to disk. Consider for example the situation illustrated in Figure 2 where a tree consists of four leaves. The leaves 1, 2 and 4 are written out to disk, whereas the tuples of leaf 3 remain in memory.

Next, we process all tupels of S one by one, see Figure 3. We transform each tupel of S into a query which is directed by the index to each leaf or Rpartition that may contain join partners. If a leaf is empty, that means its Rpartition resides on disk, we add the query to its Spartition. Note that depending on the type of the join, the tupels of S may be replicated here. If the leaf is not empty, we report those of its tupels which satisfy the query predicate.

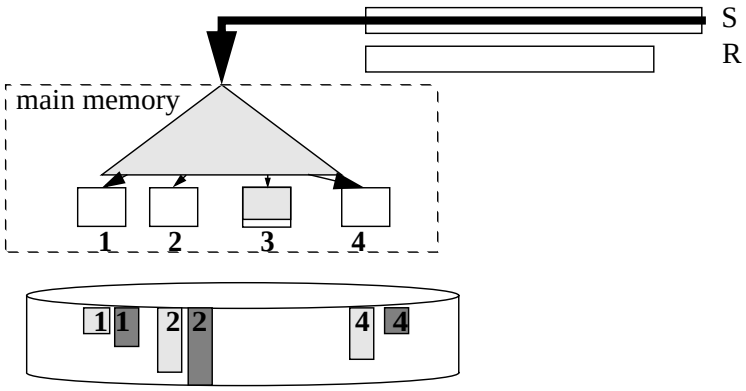

Figure 3: Processing the outer relation

When all queries are processed and the Spartitions are written to disk, we have generated pairs of partitions on disk where the one partition belongs to relation R and the other to S. The memory resident index is deleted since it is not needed anymore. After that, we apply the same algorithm in a recursive fashion to each of these pairs of partitions.

3 Use-cases of Plug&Join

In this section we describe in more detail how to apply our generic algorithm to specific use-cases. For sake of simplicity, we first start with the traditional equi join. Thereafter, we present the implementation of the spatial join and the subset join as special cases of Plug&Join. Most important in the following discussion is an appropriate choice of an index structure. Recall that Plug&Join uses the same type of index structure for partitioning the relations as well as for processing an in-memory index nested-loops join.

The GiST frameworks allows to implement with little effort all index structures that are discussed in this chapter, because all of them belong to the class of grow-and-post trees. As a consequence, implementing a specific index structure results in an implementation of a variety of join types that can be computed by Plug&Join using that index structure.

3.1 Equi Join

Let us assume that the join predicate is defined on attributes R.A and S.B of R and S, respectively. The equi join consists of the cartesian product of R and S where R.A = S.B. Plug&Join requires an index structure which efficiently supports insertions and exact match queries. These requirement is fulfilled by many different types of index structures including AVL-trees and red-black trees. We however expect that these main memory index structures are not available in current DBMS, and furthermore, they do not belong to the class of grow-and-post trees. Since B+-trees are the standard index structure in a DBMS for such a purpose, we also choose it to serve as the underlying index structure of Plug&Join.

Plug&Join creates only memory-resident indices. The index therefore has to be initialized such that the CPU-time of the relevant operations is minimized. As a general design goal, the maximum fan-out of the internal nodes and the capacity of the leaves of the index should be small enough: for B+-trees, small nodes will result in CPU-time savings when tupels of the inner relation R are inserted, whereas the processing of tupels of the outer relations S is not influenced by the size of the nodes because it is possible to use binary search in the nodes of a B+-tree.

While the tuples of an inner partition are inserted into the B+-tree, the tuples of the corresponding outer partition are transformed into queries. In case of processing equi joins, the tuples are interpreted as exact match queries which are processed on B+-trees. As a consequence, the tuples of the outer relation are assigned to at most one of the outer partitions.

The equi join is a symmetric join in that both relations may serve as the inner relation which serves as the source for creating the indices. This gives us the possibility to enhance our algorithm in the following way. For each invocation of our algorithm we declare the smaller partition to become the inner one. In general, this will reduce the number of recursive calls.

We previously emphasized that Plug&Join in combination with the same index structure can even support different types of joins, the following explains this more clearly. B+-tree supports primarily two types of queries: exact match query and range query. The so-called band join [DNS 91] is an example where the outer relation has to be transformed into range queries. This is however the only difference to the equi join when Plug&Join would be used to implement a band join.

3.2 Spatial Join

For the spatial join, the input relations R and S refer to a set of spatial objects. The result of the spatial join of R and S is defined to be the subset of the cartesian product of R and S where the spatial predicate is satisfied. In common with most work on spatial join processing, we assume here that the join predicate tests whether spatial objects overlap and that the spatial objects are rectilinear rectangles. In general, these rectangles refer to the minimum bounding boxes (MBB) of the actual objects. They can be used in a filter step [Ore 86] to prune the search space and to deliver a candidate set of the spatial join. An additional refinement step is required to identify the candidates which really satisfy the join predicate.

In order to use Plug&Join for spatial joins, an efficient index structure is required that supports insertions of objects (without clipping them) and window queries. The R*-tree [BKSS 90], a variant of the R-tree [Gut 84], is among the most efficient index structures which fulfill these requirements. The R*-tree groups

spatial objects hierarchically by their MBBs. The leaves of the R-tree contain the spatial objects (or references to them), whereas an entry of an internal node consists of the MBB of the spatial objects stored in its associated subtree. Due to the lack of ordering rectangles, the CPU-time consumption is generally higher than for a B+-tree. Splitting a node in the R*-tree requires O(d*B log B) time where B denotes the capacity of a node and d is the number of dimensions (here d=2). A window query also requires that all entries of a visited node have to be examined. It is therefore even more important than in case of B+-trees to set the maximum fan-out and the capacity of the leaf nodes to small values. Otherwise, the required CPU-time to insert and retrieve tuples is much too high leading to an inefficient implementation of the spatial join. We will show in our experiments in section 4.3 how to choose these two parameters.

For the spatial join supporting the overlap join predicate, the objects of the outer relation are transformed into window queries. In general, several leaves are required to answer window queries and therefore, objects of the outer relation are replicated by Plug&Join. However, the impact of replication on the performance is rather small for non-artificial data sets, because the volume of an object is small compared to the volume of the minimum bounding box of a partition. Moreover, the R*-tree has been designed to minimize the overlap of the bounding boxes stored in a node. This also results in a low replication of the outer relation.

Similar to the equi join, the spatial join is also a symmetric join such that both relations may serve as the inner relation. For each call of Plug&Join we always choose the smaller of both partitions to serve as the inner one.

3.3 Subset Join

In contrast to the pure relational data model, object-oriented models like the ODMG model [Cat 96] support set-valued attributes. As a consequence, it might be possible to specify joins based on set comparisons, for example whether the one set-value is a subset of the other. The subset join is of practical relevance because it implements a sort of a *forall*-predicate. As an example, consider relations *students* and

professors where a set-valued attribute *courses* exists in both of the relations. For each professor, we might be interested in those students which attended all courses of this professor. The corresponding query can be expressed as a subset join.

In order to support subset joins, we need an index structure that support insertions and subset queries. Let R and S be relations with set-valued attributes R.A and S.B, respectively. The subset join computes the subset of the cartesian product of R and S where $R.A \supseteq S.B$ is satisfied. Among the most efficient index structures to support subset queries is the S-tree [Dep86]. S-trees are grow-and-post trees which can be implemented under GiST. S-trees use signatures as a kind of approximation for a set-valued object where a signature is a bit vector of a predefined length. The leaves of the S-tree contain signatures of the objects, whereas an entry of an internal node of the S-tree consists of the signature which is computed by a bitwise OR on the bit vectors of all objects stored in the corresponding subtree. In our example, the bit vector of a student is computed by a bitwise OR on the bit vectors of the courses he/she attended.

A subset query is performed in the following way. First, the query is transformed into a bit vector. In our example, each professor corresponds to a single query, and the bit vector is obtained analogously to students. Then, the query starts at the root of the S-tree and examines the bit vectors of each entry. If the bit vector of S is part of the bit vector of an entry, the corresponding subtree has to be examined recursively. Similar to the R-tree, the S-tree only provides candidates. Therefore, in a separate step the candidates must be tested whether they really fulfill the query condition. For the details of the search algorithm we refer the reader to [Dep 86].

Due to the overhead of splitting and searching nodes, we also suggest to choose the capacity of a leaf node and the fan-out of an internal node as small as possible. Moreover, a query generally proceeds to multiple leaves and therefore, replication occurs for the outer relation. In contrast to spatial joins and equi joins, the subset join is not symmetric, so that we are not able to swap the roles of the inner and outer partition on demand.

Despite the large amount of work in the area of join processing, we are not aware of a description of algo-

rithms to process large subset joins. Note that [HM 97] presented several proposals, but input relations are assumed to be small enough to process the entire join in main memory.

4 Experiments

In this section we present a few results from our experiments with Plug&Join. In particular, we show the results of a comparison of Plug&Join and a special-purpose method. In our experiments, we decided to examine the spatial join for the following reasons: First, the spatial join is the best-known non-equi join and many special-purpose algorithms have been proposed recently. Second, large real databases are available which can be used as data sources in our experiments. Among the different algorithms for spatial joins we choose the Partition Based Spatial-Merge Join (PBSM) [PD 96] as the competitor of Plug&Join. PBSM has been shown in different experiments to be among the most efficient join methods ([KS 97], [APR+ 98], [DS 99], [MP 99]).

In the following we first give a detailed description of PBSM. In our experiments a modified version of PBSM [DS 99] is used which runs considerably faster than the original method. This is followed by an introduction of our computing model and a description of the spatial data sets. Thereafter, we first discuss some of the results which justify our claims previously made for Plug&Join. Finally, we present the results of our comparison of PBSM and Plug&Join.

4.1 PBSM

Partition Based Spatial-Merge Join (PBSM) [PD 96] is a divide & conquer algorithm that breaks up the input relations R and S into partitions using an equidistant grid. Similar to Plug&Join it is then sufficient to compute the join for pairs of partitions (where one belongs to R and the other belongs to S). We assume that the input relations refer to sets of key-pointer elements (KPE). A KPE consists of a pointer to a spatial object and its MBB.

PBSM as it was originally proposed in [PD 96] performs in four phases which are described briefly in the

following. In the first phase, the number of partitions *p* is computed such that the join of a pair of partitions can be processed in main memory (with high probability). For each of the input relations, *p* partitions are created by using an equidistant grid with *NT* cells, $NT \geq p$. A cell of the grid, also termed *tile*, is then assigned to one of the partitions. A KPE of a relation is inserted into a partition of the relation if its MBB intersects with one of the tiles that belong to the partition. This rule obviously results in replication of KPEs, i. e. a KPE can be stored in more than one partition. The advantage of assigning multiple tiles to a partition is that the KPEs are almost uniformly distributed among the partitions. Patel and DeWitt [PD 96] recommend to use a hash function for mapping the tiles to partitions.

Let $R_1,\ldots,R_p$ and $S_1,\ldots,S_p$ be the partitions of the relations R and S, respectively. In the second phase, pairs (R_i, S_i) of partitions are treated which do not fit into main memory. For these pairs of partitions, repartitioning has to be performed in a recursive fashion. In the third phase, a corresponding pair of partitions is loaded into main memory and an in-memory join is performed using the plane-sweep method originally presented in [BKS 93]. Note that due to the replication of KPEs in different partitions the same result can be produced more than once. In order to eliminate these duplicates in the response set, the results obtained from the three phases are sorted in a final phase.

An important improvement of PBSM is presented in [DS 99]. Instead of eliminating duplicates in a separate phase (which has a few serious disadvantages) an inexpensive on-line algorithm is proposed to decide whether a result of the first phase is a duplicate or not. Experiments in [DS 99] have shown that the new method for duplicate elimination results in a substantial improvement over the original approach. Therefore, we also used PBSM with the new method for duplicate elimination in our experiments.

Important to PBSM is a good choice for p (number of partitions) and NT (number of tiles). Let M be the size of the available main memory and let sizeof(KPE) be the size of a KPE (36 bytes in our experiments). We suggest the following formula for computing p:

$$p = \left\lceil sf \times \frac{(\|R\| + \|S\|) \cdot sizeof(KPE)}{M} \right\rceil \qquad \text{Formula 1}$$

where $sf \geq 1$ is the scaling factor. We observed that for $sf = 1$ as it was originally proposed the size of a

pair of partitions is frequently larger than M and therefore, repartitioning has to be performed too often. In our experiments we found that *sf* = 1.5 is the best choice. The assignment of tiles to partitions is performed in a round-robin fashion (row by row). In order to avoid homogeneous assignment patterns NT was chosen such that p is not a divisor of NT. The parameter NT was almost 60 times larger than p in our experiments.

4.2 The Computing Model

In our I/O model, data is transferred between main memory and secondary storage in pages of fixed size. The cost for reading a page consists of positioning the disk arm and transferring the page. Moreover, we also assume that a *contiguous* sequence of multiple pages can be read (written) with positioning the disk arm only once. Such a multi-page read is obviously less expensive than reading each of the pages separately. Our implementations were designed in such a way that multi-page I/Os are used whenever possible.

In the following, we investigate the performance of spatial join algorithms in different experiments using real data sets. The performance of the algorithms is simply measured by the total runtime (in seconds) of the corresponding C++ implementations on an Intel PC (with a Celeron 333 MHZ processor, 128 MB main memory and a 4 GB Mikropolis disk) under Windows 95. Our current implementations do not support an overlapped processing of I/O and CPU. The join algorithms were allowed to use only a fraction of main memory, whereas the buffers of the operating system were turned off throughout our experiments.

data set	description	number of MBBs	coverage
CAL_RR	railways and rivers in California	625,640	0.21
CAL_ST	streets in California	1,888,012	0.12

Table 1: Datasets used in the experiments

In our experiments we used different real data sets derived from the TIGER files [Bur 96]. Due to space limitations we are only able to present the results of our largest spatial join in detail. We also performed other joins whose results are in agreement with the ones we present in this paper. The first data set CAL_RR contains the MBBs of the railways, rivers, administrative and hydrographic features from the state California, whereas the second data set CAL_ST contains the MBBs of the streets. The coverage of

the data files (which is defined by the sum of the area of the MBBs divided by the area of the MBB of all MBBs) is rather small for these files. This is generally true for most of the real data sets. In Table 2 the output parameters of the spatial join are reported. The selectivity refers to the number of results divided by the product of the number of MBBs of the input relations.

join	R	S	number of results	selectivity
J1	CAL_RR	CAL_ST	1,683,888	1.43×10^{-6}

Table 2: Spatial joins performed in our experiments

4.3 Plug&Join

In this section, we discuss how to choose the required parameters for Plug&Join. The generic algorithm requires appropriate values for the leaf capacity of the index and the fan-out of the internal nodes. In Figure 4, we present the execution time of spatial joins for different settings of the parameters. Plug&Join

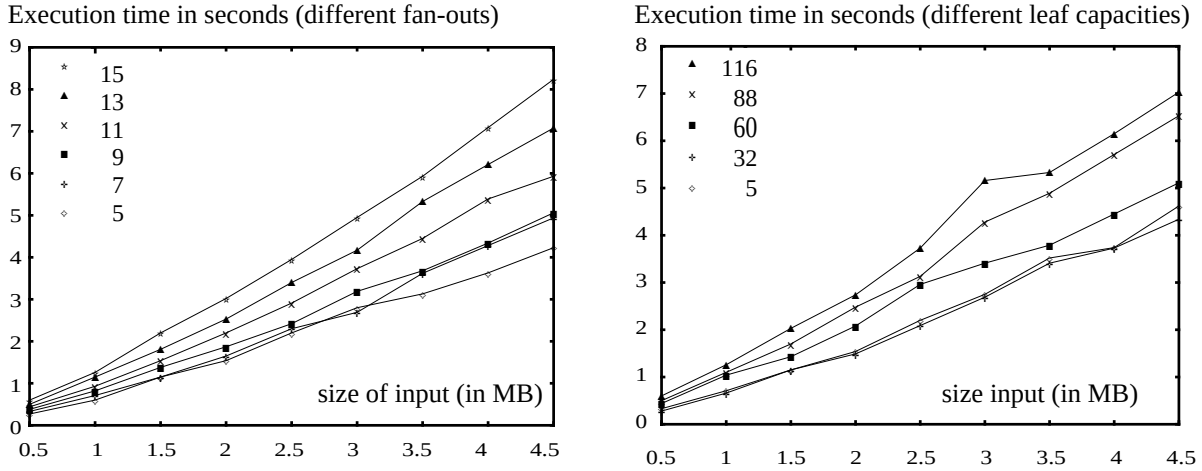

Figure 4: Execution time as a function of the size of the input relations

employs the R*-tree in these experiments. In order to present the results as a function of the size of the partitions, we limit the join to the result set of window queries of different sizes which were performed on relations CAL_ST and CAL_RR. We examine in the following the case where the main memory is large enough to keep the input of the spatial join.

On the left-hand side of Figure 4, we plot the curves for different fan-outs of the internal pages of the index. The results show that small fan-outs are most promising. This result is not evident since small fan-

outs reduces the quality of the partitioning of the R*-tree and therefore, they increase the cost of the window queries. This is because an insertion has fewer choices for proceeding from a node to one of its siblings. However, small fan-outs also reduce the cost of insertions. These performance gains are more relevant than the performance loss for the queries.

On the right-hand side of Figure 4, the execution time is plotted for different settings of the leaf capacity. The results show that the size of the leaves should be also small. There is no remarkable performance difference between a leaf capacity of 5 and 32. We choose in the following the larger value since the size of the index will be considerably lower for this value.

4.4 Comparison of PBSM and Plug&Join

In this section we present the execution time of PBSM and Plug&Join when the spatial join J1 is processed. Both methods are using formula 1 to compute the number of partitions build in each partitioning step. Plug&Join uses the R*-tree where the leaf capacity is 32 and the fan-out of the internal nodes is 5. In Figure 5, we plotted the execution time of both join algorithms as a function of the available main memory.

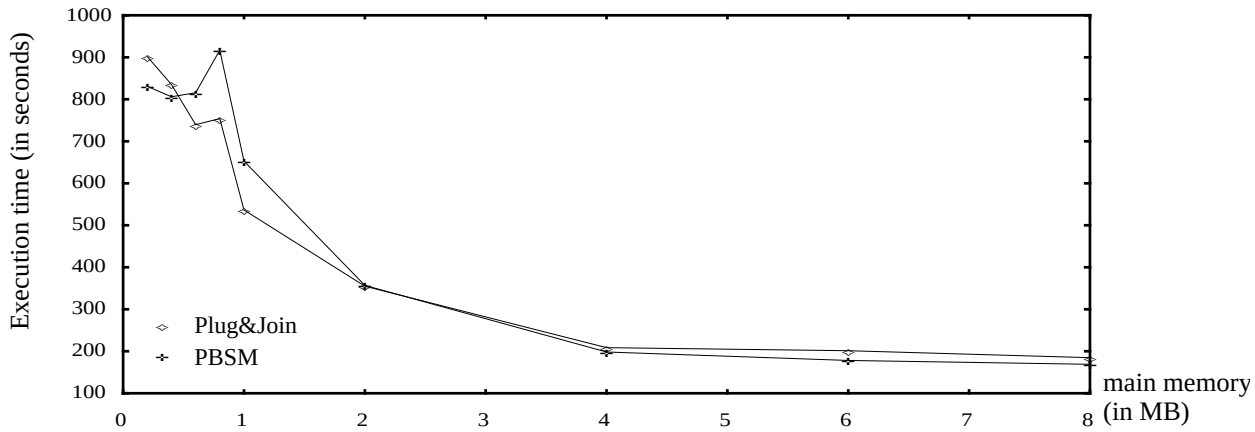

Figure 5: Execution time as a function of the size of main memory for Plug&Join and PBSM

Plug&Join seems to be slightly superior for medium-sized memory, whereas PBSM performs slightly better for large main memories. Overall, these results basically show that both methods give similar performance.

5 Conclusions

This paper presents Plug&Join, a generic algorithm particularly suitable for processing non-equi joins in an extensible database system. Plug&Join requires an index structure satisfying the grow-and-post interface [Lom 91] for its initialization, but a preexisting index on any of the relations is not required. The index structure is used for two different problems: First, an in-memory index is build-up for partitioning both of the relations. Partitioning employs the ability of the index structure to support insertions and queries where the query predicate is directly derived from the join predicate. Second, temporarily built in-memory indices are used for processing joins on partitions that fit in memory.

The benefit of Plug&Join is that it can be easily customized to different types of joins. There are three possible approaches to customizing: First, the most easiest would be when the required index structure is already available in the underlying system and the desired query predicate is supported. Second, the index structure might be implemented, but the query predicate has to be provided. Third, the index structure has to be implemented completely. Fortunately, generic toolboxes for implementing index structures like GiST [HNP 95] can be used to simplify the last two approaches. In this paper, we present different use-cases for processing equi joins, spatial joins and subset joins. The use-case for subset joins presents the first algorithm, we are aware of, that efficiently treats the subset predicate for large relations. For the spatial join, we present first results of a preliminary performance comparison of Plug&Join (using the R*-tree) with PBSM [PD 96], one of the most efficient special-purpose methods. Overall, our results show that Plug&Join is competitive to PBSM.

The general question we address in this paper is whether special-purpose approaches are really necessary for supporting new types of non-equi joins. The results of our paper give a strong indication that such special-purpose approaches are not very effective. We believe that their design does not meet the demands of a modern DBMS.

In our future work we are interested in further optimization of our generic approach. One desirable optimization might be to incorporate the well-known hybrid join technique. Note that all of our use-cases will

benefit from such an optimization. Furthermore, we continue with the implementations of the use-cases which will be part of a powerful generic query processing library.

References

[Aok 98] Aoki, P. M.: Generalizing "Search" in Generalized Search Trees (Extended Abstract). ICDE 1998: 380-389

[APR+ 98] Arge, L.; Procopiuc, O.; Ramaswamy, S.; Suel, T.; Vitter, J. S.: Scalable Sweeping-Based Spatial Join. VLDB 1998: 570-581

[BFH 93] Becker, L.; Finke, U.; Hinrichs, K.: A New Algorithm for Computing Joins with Grid Files. ICDE 1993: 190-197

[BGO+ 96] Becker, B.; Gschwind, S.; Ohler, T.; Seeger, B.; Widmayer, P.: An Asymptotically Optimal Multiversion B-Tree. VLDB Journal 5(4): 264-275 (1996)

[BKS 93] Brinkhoff, T.; Kriegel, H.-P.; Seeger, B.: Efficient Processing of Spatial Joins Using R-Trees. SIGMOD Conference 1993: 237-246

[BKSS 90] Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B.: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference 1990: 322-331

[BSW 99] Van den Bercken, J.; Seeger, B.; Widmayer, P.: The Bulk Index Join: A Generic Approach to Processing Non-Equijoins. ICDE 1999: 257

[Bur 96] Bureau of the Census: Tiger/Line Precensus Files: 1995 technical documentation. Bureau of the Census, Washington DC. 1996

[Cat 96] Cattell, R. (editor): The Object Database Standard: ODMG-93, Release 1.2, Morgan Kaufmann, 1996

[CPZ 97] Ciaccia, P.; Patella, M.; Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. VLDB 1997: 426-435

[Dep 86] Deppisch, U.: S-tree: A Dynamic Balanced Signature Index for Office Retrieval. SIGIR 1986: 77-87

[DNS 91] DeWitt, D. J.; Naughton, J. F.; Schneider, D. A.: An Evaluation of Non-Equijoin Algorithms. VLDB 1991: 443-452.

[DS 99] Dittrich, J.; Seeger, B.: Data Redundancy and Duplicate Detection in Spatial Join Processing, Technical Report 18, Department of Mathematics and Computer Science, Univ. Marburg, 1999.

[Gra 93] Graefe, G.: Query Evaluation Techniques for Large Databases. Computing Surveys 25(2): 73-170 (1993)

[Gün 93] Günther, O.: Efficient Computation of Spatial Joins. ICDE 1993: 50-59

[Gut 84] Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Conference 1984: 47-57

[HNP 95] Hellerstein, J. M.; Naughton, J. F.; Pfeffer, A.: Generalized Search Trees for Database Systems. VLDB 1995: 562-573

[HM 97] Helmer, S.; Moerkotte, G.: Evaluation of Main Memory Join Algorithms for Joins with Set Comparison Join Predicates. VLDB 1997: 386-395

[HS 95] Hoel, E. G.; Samet, H.: Benchmarking Spatial Join Operations with Spatial Output. VLDB 1995: 606-618

[HSW 89] Henrich, A.; Six, H.-W.; Widmayer, P.: The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. VLDB 1989: 45-53

[Lom 91] Lomet, D. B.: Grow and Post Index Trees: Roles, Techniques and Future Potential. SSD 1991: 183-206.

[LR 94] Lo, M.-L.; Ravishankar, C. V.: Spatial Joins Using Seeded Trees. SIGMOD Conference 1994: 209-220

[LR 96] Lo, M.-L.; Ravishankar, C. V.: Spatial Hash-Joins. SIGMOD Conference 1996: 247-258

[KS 97] Koudas, N.; Sevcik, K. C: Size Separation Spatial Join. SIGMOD Conference 1997: 324-335

[ME 92] Mishra, P.; Eich, M. H.: Join Processing in Relational Databases. Computing Surveys 24(1): 63-113 (1992)

[MP 99] Mamoulis, N.; Papadias, D.: Integration of Spatial Join Algorithms for Processing Multiple Inputs. SIGMOD Conference 1999: 1-12.

[Ore 86] Orenstein, J.: Spatial Query Processing in an Object-Oriented Database System. SIGMOD Conference 1986: 326-336

[PD 96] Patel, J. M.; DeWitt, D. J.: Partition Based Spatial-Merge Join. SIGMOD Conference 1996: 259-270

[PTSE 95] Papadias, D.; Theodoridis, Y.; Sellis, T. K.; Egenhofer, M. J.: Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. SIGMOD Conference 1995: 92-103

[Sha 86] Shapiro, L. D.: Join Processing in Database Systems with Large Main Memories. TODS 11(3): 239-264 (1986)

[SSJ 94] Soo, M. D.; Snodgrass, R. T.; Jensen, C. S.: Efficient Evaluation of the Valid-Time Natural Join. ICDE 1994: 282-292

[Zur 97] Zurek, T.: Optimisation of Partitioned Temporal Joins. BNCOD 1997: 101-115