A Performance Evaluation of Spatial Join Processing Strategies*

Apostolos Papadopoulos[1], Philippe Rigaux[2], Michel Scholl[2]

[1] Data Engineering Lab., Aristotle Univ., 54006 Thessaloniki, Greece,
[2] Cedric/CNAM, 292 rue St Martin, F-75141 Paris Cedex 03, France

Abstract. We provide an evaluation of query execution plans (QEP) in the case of queries with one or two spatial joins. The QEPs assume R*-tree indexed relations and use a common set of spatial joins algorithms, among which one is a novel extension of a strategy based on an on-the-fly index creation prior to the join with another indexed relation. A common platform is used on which a set of spatial access methods and join algorithms are available. The QEPs are implemented with a general iterator-based spatial query processor, allowing for pipelined QEP execution, thus minimizing memory space required for intermediate results.

1 Introduction

It is well known that the application of Database Management Systems (DBMS) join techniques, such as sort-merge, scan and index, hash join and join indices, to the context of spatial data is not straightforward. This is due to the fact that these techniques, as well as B-tree-based techniques, intensively rely on the domain ordering of the relational attributes, which ordering does not exist in the case of multi-dimensional data.

A large number of spatial access methods (SAM) have been proposed in the past fifteen years [VG98] as well as a number of spatial join algorithms [Ore86,GS87,Gun93,BKS93,LR96,PD96,HJR97,APR+98], some of them relying on the adaptation of well-known join strategies to the particular requirements of spatial joins.

These strategies have been validated through experiments on different platforms, with various methodologies, datasets and implementation choices. The lack of a commonly shared performance methodology and benchmarking renders difficult a fair comparison among these numerous techniques.

The methodology and evaluation are crucial not only for the choice of a few efficient spatial join algorithms but also for the optimization of complex queries involving several joins in sequence (multi-way joins). In the latter more general case, the generation and evaluation of complex query execution plans (QEP) is

* Work supported by the European Union's TMR program ("Chorochronos" project, contract number ERBFMRX-CT96-0056) and by the 1998-1999 French-Greek bilateral protocol.

central to optimization. Only a few papers study the systematic optimization of spatial queries containing multi-way joins [MP99].

The objective of this paper is two fold: (i) to provide a common framework and evaluation platform for spatial query processing, and (ii) to use it to experimentally evaluate spatial join processing strategies.

A complex spatial query can be translated into a QEP with some physical operations such as data access (sequential or through an index), spatial selection, spatial join, sorting, etc. A QEP is then represented as a binary tree in which leaves are either indices or data files and internal nodes are physical operators.

We use as a model for spatial query processing the pipelined execution of such QEPs with each node (operation) being implemented as an *iterator* [Gra93]. This execution model provides a sound framework: it encompasses spatial and non-spatial queries, and allows to consider in an uniform setting simple and large complex queries involving several consecutive joins. Whenever possible, records are processed one-at-a-time and transfered from one node to the following, thereby avoiding the storage of intermediate results.

Such an execution model is not only useful to represent and evaluate complex queries, but also to specify and make a fair comparison of simple ones. Indeed, consider a query including a single spatial join between two relations. The join output is, unfortunately, algorithm dependent. Some algorithms provide as an output a set of pairs of record identifiers (one per relation), others, such as the so-called *Scan and Index* (SAI) strategy provide a set in which each element is composed of a record (of the first relation) and the identifier of a record in the second relation. Then to complete the join, the former case requires two data accesses, while only one data access is necessary in the latter case. This example illustrates the necessity for a consistent comparison framework. The above execution model provides such a framework.

Another advantage of this execution model is that it allows not only to compare two QEPs on their time performance but also on their memory space requirement. Some operations cannot be pipelined, e.g., sorting an intermediate result, and require the completion of an operation before starting the following operation. Such operators, denoted *blocking iterators* in this paper, are usually memory-demanding and raise some complex issues related to the allocation of the available memory among the nodes of a QEP. In order to make a fair comparison between several QEPs, we shall always assign the *same* amount of memory to each QEP during an experiment.

The study performed in this paper is a first contribution to the evaluation of complex spatial queries that may involve several joins in sequence (multi-way joins). Based on the above model we evaluate queries with one or two spatial joins. We make the following assumptions: (i) all relations in the database are indexed on their spatial attribute, (ii) we choose the R*-tree [BKSS90] for all indices, (iii) the index is always used for query optimization. While the first assumption is natural, the second one is restrictive. Indeed, while the R*-tree is an efficient SAM, there exists a number of other data structures that deserve some attention, among which it is worth noting the grid based structures derived

from the grid file [NHS84]. The third assumption is also restrictive since it does not take into account the proposal of several techniques for joining non indexed relations.

The comparison of QEPs as defined above has been done on a common general platform developed for spatial query processing evaluation. This platform provides basic I/O and buffer management, a set of representative SAMs, a library of spatial operations, and implements a spatial query processor according to the above iterator model using as nodes the SAMs and spatial operations available.

The rest of the paper is organized as follows. Section 2 briefly surveys the various spatial join techniques proposed in the literature and summarizes related work. The detailed architecture of the query processor is presented in Section 3. Section 4 deals with our choices for spatial join processing and the generated QEPs. Section 5 reports on the experiment, the datasets chosen for the evaluation and the results of the performance evaluation. Some concluding remarks are given in Section 6.

2 Background and Related Work

We assume each relation has a spatial attribute. The spatial join between relations R_1 and R_2 constructs the pairs of tuples from $R_1 \times R_2$ whose spatial attributes satisfy a spatial predicate. We shall restrict this study to *intersect joins*, also referred to as *overlap* joins. Usually, each spatial attribute has for a value a pair (*MBR*, *spatial object representation*), where *MBR* is the minimum bounding rectangle of the spatial object. Intersect spatial joins are usually computed in two steps. In the *filter step* the tuples whose MBR overlap are selected. For each pair that passes the filter step, in the *refinement step* the spatial object representations are retrieved and the spatial predicate is checked on these spatial representations [BKSS94].

Many experiments only consider the filter step. This might be misleading for the following reasons: first one cannot fairly compare two algorithms which do not yield the same result (for instance if the SAI strategy is used, at the end of the filter step, part of the record value has been already accessed, which is useful for the refinement step, while it is not true with the STT strategy [Gun93,BKS93]), second by considering only the filter step, one ignores its interactions with the refinement step, for instance in terms of memory requirements. We shall include in our experiments all the operations necessary to retrieve data from disk, whether this data access is for the filter step, or the refinement step. Only the evaluation of the computational geometry algorithm on the exact spatial representation, which is equivalent whatever the join strategy, will be excluded.

SAMs can be roughly classified into two categories:

- *Space driven* structures, among which *grids* and *quadtrees* are very popular, partition the tuples according to some spatial scheme independent from the spatial data distribution of the indexed relation.

– *Data-driven* structures, on the other hand, adapt to the spatial data distribution of tuples. The most popular SAM of this category is the R-tree [Gut84]. The R^+-tree [SRF87], the R^*-tree [BKSS90] and the X-tree [BKK96] are improved versions of the R-tree. These *dynamic* SAMs maintain their structure on each insertion/deletion. In the case of *static* collections which are not often updated, packing algorithms [RL85,KF93,LEL97] build optimal R-trees, called packed R-trees.

Spatial joins algorithms can be classified into three categories depending on whether each relation is indexed or not.

1. *no index*: for the case where no index exists on any relation, several *partitioning* techniques have been proposed which partition the tuples into buckets and then use either hashed based techniques or sweep-line techniques [GS87,PD96,LR96,KS97,APR+98].
2. *two indices*: when both relations are indexed, the algorithms that have been proposed depend on the SAM used. [Ore86] is the first known work on spatial joins. It proposes a 1-dimensional ordering of spatial objects, which are then indexed on their rank in a B-tree and merge-joined. [Gun93] was the first proposal of an algorithm called Synchronized Tree Traversal (STT) which adapts to a large family of spatial predicates and tree structures. The STT algorithm of [BKS93] is the most popular one because of its efficiency. Proposed independently from [Gun93], it uses R^*-trees and an efficient depth-first tree traversal of both trees for intersection joins. The algorithm is sketched below.

   Algorithm STT (Node N_1, Node N_2)
   begin
       for all (e_1 in N_1)
           for all (e_2 in N_2) such that $e_1.MBR \cap e_2.MBR \neq \emptyset$
               if (the leaf level is reached) **then**
                   output (e_1, e_2)
               else
                   N_1' = readPage ($e_1.pageID$); N_2' = readPage ($e_2.pageID$);
                   STT(N'1, N'2)
               endif
   end

   Advanced variants of the algorithm apply some local optimization in order to reduce the CPU and I/O costs. In particular, when joining two nodes, the overlapping of entries is computed using a plane-sweeping technique instead of the brute-force nested loop algorithm shown above. The MBRs of each node are sorted on the x-coordinate, and a merge-like algorithm is carried out. This is shown to significantly reduce the number of intersection tests.
3. *single index* : when only one index exists, the simplest strategy is the Scan And Index (SAI) strategy, a variant of the nested loop algorithm which scans the non-indexed relation and for each tuple r delivers to the index of the other

relation a window query with $r.MBR$ as an argument. The high efficiency of the STT algorithm suggests that an "on-the-fly" construction of a second index, followed by STT, could compete with SAI. This idea has inspired the join algorithm of [LR98] which constructs a *seeded-tree* on the non indexed relation which is a R-tree whose first levels match exactly those of the existing R-tree. It is shown that the strategy outperforms SAI and the naive on-the-fly construction of an R-tree with dynamic insertion. An improvement of this idea is the SISJ algorithm of [MP99]. An alternative is to build a *packed R-tree* by using bulk-load insertions [LEL97]. Such constructions optimize the STT algorithm since they reduce the set of nodes to be compared during traversal. These algorithms are examples of strategies referred to as *Build-and-Match* strategies in the sequel.

The complexity of the spatial join operation, the variety of techniques and the numerous parameters involved in a spatial join render extremely difficult the comparison between the above proposals briefly sketched above.

Only a few attempts have been made toward a systematic comparison of spatial join strategies. [GOP+98] is a preliminary attempt to integrate in a common platform the evaluation of spatial query processing strategies. It proposes a web-based rectangle generator and gives first results on the comparison of three join strategies: nested loop, SAI and STT. The major limit of this experiment is that it is built on top of an existing DBMS. This not only limits the robustness of the results but renders impossible or inefficient the implementation of complex strategies, the tuning of numerous parameters and a precise analysis.

[MP99] is the first study on multi-way spatial joins. It proposes an iterator pipelined execution of QEPs [Gra93] for multi-way spatial joins, with three join algorithms, one per category, namely STT, SISJ, and a spatial hash-join technique [LR96]. An analytical model predicts the cost of QEPs, a dynamic programming algorithm for choosing the optimal QEP is proposed. The query optimization model is validated through experimental evaluation.

The modeling of QEPs involving one or several joins in the study reported below follows the same pipelined iterator based approach. This execution model is implemented on a platform common to all evaluations. This platform allows for fine tuning of parameters impacting the strategies performance and is general enough to implement and evaluate any complex QEP: it is not limited to spatial joins. Last but not least, such a model and its implementation allow for various implementation details generally absent from evaluations reported in the literature. The relation access after a join is an example of implementation "detail" which accounts for an extremely significant part of the query response time as shown below.

3 The Query Processor

The platform has been implemented in C++ and runs on top of UNIX or WindowsNT. Its architecture is shown in Fig. 1. It is composed of a *database* and

three modules which implement some of the standard low-level services of a centralized DBMS.

The *database* is a set of binary files. Each binary file either stores a *data file* or a SAM. A data file is a sequential list of records. A SAM or index refers to records in an indexed data file through *record identifiers*. The lowest level module is the I/O module, which is in charge of reading (writing) pages from (to) the disk. The second module manages buffers of pages fetched (flushed) from (to) the disk through the I/O module. On-top of the buffer management module is the query processing module, which supports spatial queries.

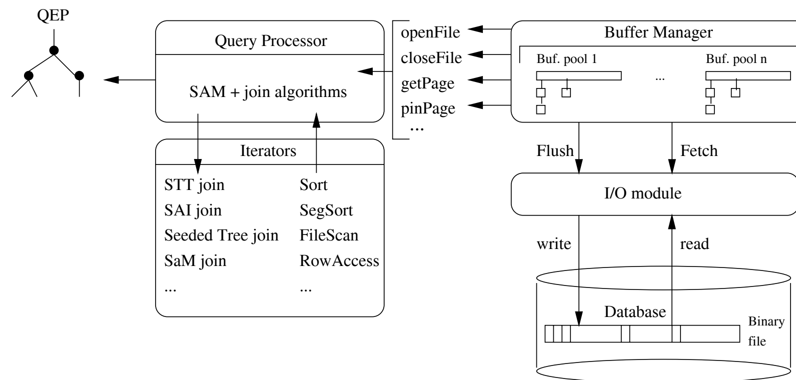


Fig. 1. Platform architecture

Database and Disk Access

The database (data files and SAM) is stored in binary files divided into *pages* whose size is chosen at database creation. A page is structured as a *header* followed by an array of fixed-size records which can be either data records or *index entries*. The header and record sizes depend on the file. By knowing the record size, one can compute the number of records per page and the data file size.

Each page is uniquely identified by its `PageID` (4 bytes). A record is identified by a `RecordID` (8 bytes) which is a pair `[PageID, offset]` where `offset` denotes the record offset in the page.

1. *Data files* are sequential collections of pages storing *data records*. In the current setting, a record basically is a binary representation of a spatial object. From the query processing point of view, the most important information stored in a record is its *geometric key*, which is, throughout this experiment, its MBR. A data file can either be accessed sequentially (*FileScan* in the sequel) , or by `RecordID` (*RowAccess* in the sequel). It is important to note that the datafiles are *not* clustered on their geometric representation (i.e., objects close in space are not necessarily close on disk).

2. *SAMs* are structured collections of `IndexEntry`. An index entry is a pair `[Key, RecordID]`, where `Key` denotes the geometric key (here the MBR) and `RecordID` identifies a record in the indexed data file. The currently implemented SAMs are a grid file, an R-tree, an R*-tree and several packed R-trees. In the sequel, each datafile is indexed with an R*-tree.

Buffer management

The buffer manager handles one or several *buffer pools*: a data file or index (SAM) is assigned to one buffer pool, but a buffer pool can handle several indices. This allows much flexibility when assigning memory to the different parts of a query execution plan. The buffer pool is a constant-size cache with LRU or FIFO replacement policy (LRU by default). Pages can be *pinned* in memory. A pinned page is never flushed until it is unpinned.

Currently, all algorithms requiring page accesses uniformly access these pages through the interface provided by the buffer manager. In particular, spatial join algorithms share this module and therefore cannot rely on a tailored main memory management or a specialized I/O's policy unless it has already been implemented in this module.

Query processing module

One of the important design choices for the platform is to allow for any experimental evaluation of *query execution plans* (QEP) as generated by database query optimizers with an algebraic view of query languages. During optimization, a query is transformed into a QEP represented as a binary tree which captures the order in which a sequence of *physical* algebraic operations are going to be executed. The leaves represent data files or indices, internal nodes represent algebraic operations and edges represent dataflows between operations. Examples of algebraic operations include data access (*FileScan* or *RowAccess*), spatial selections, spatial joins, etc. As mentioned above we use as a common framework for query execution, a demand-driven process with iterator functions [Gra93]. Each node (operation) is an iterator. This allows for a pipelined execution of multiple operations, thereby minimizing the system resources (memory space) required for intermediate results: data consumed by an iterator, say I, is generated by its son(s) iterator(s), say J. Records are produced and consumed one-at-a-time. Iterator I asks iterator J for a record. Therefore the intermediate result of an operation is not stored in such pipelined operations except for some specific iterators called *blocking iterators*, such as sorting.

This design allows for simple QEP creation by "assembling" iterators together. Consider the QEP for a spatial join $R \bowtie S$ implemented by the simple scan-and-index (SAI) strategy (Fig. 2.a): scan R (*FileScan*); for each tuple r in R, execute a window query on index I_S with key $r.MBR$. This gives a record ID `RecordID2` [1]. Finally read the record with id `RecordID2` in S (*RowAccess*).

[1] As a matter of fact each index (leaf) access returns an `IndexEntry`, i.e., a pair `[MBR, RecordID]`. For the sake of simplicity, we do not show this MBR on the figures.

The refinement step not represented in the figure can then be performed on the exact spatial object available in `Record1` and `Record2`.

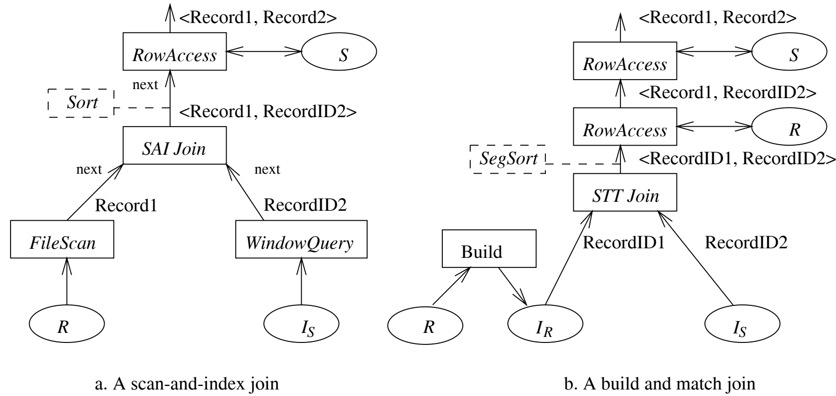


Fig. 2. Query execution plans

This is a fully pipelined QEP: therefore the response time (e.g., the time to get the first record) is minimal. It is sometimes necessary to introduce *blocking iterators* in a QEP which require the consumption of all input data before any output is possible. Then significant memory is necessary for intermediate results.

As an example, one can introduce a *Sort* blocking iterator in the QEP of Fig. 2.a in order to sort the data flow output by the SAI join on the `PageID` of the `RecordID2` component. This allows to access only once the pages of S instead of issuing random **`reads`** to get the S records, which might lead to several accesses to the same page. However no record can be delivered to the user before the join is completely processed.

As a more complex (and realistic) example, consider the QEP of Fig. 2.b. It implements a different join strategy: an index already exists on S, another one is built on the fly on R (iterator *Build*)[2], and an STT join is executed. Such a join delivers pairs of `RecordID`, hence two further *RowAccess*, one per relation, are necessary to complete the query (refinement step). *Build* is blocking: the join cannot be started before index I_R has been completely built. In addition, the maximal amount of available memory should be assigned to this iterator to avoid as much as possible to flush pages on disk during index construction.

It may happen that a QEP relies on several *blocking iterators*. In that case the management of memory is an important issue. Consider the QEP of Fig. 2.b. The STT node delivers pairs of `RecordID`, $[r_i, s_i]$, which resembles the join index upon non clustered data, as described in [Val87]. The naive strategy depicted in Fig. 2.b alternates random accesses on the datafiles R and S; then the same page (either in R or S) will be accessed several times, which leads to a large

[2] R can be an intermediate result delivered by a sub-QEP.

number of page faults. The following preprocessing algorithm is proposed in [Val87] and denoted *segmented sort* (*SegSort*) in the sequel: (1) allocate a buffer of size B; (2) compute the number n of pairs `(Record1,RecordID2)` which can fit in B; (3) load the buffer with n pairs `(RecordID1, RecordID2)`; (4) sort on `RecordID1`; (5) access relation R and load records from R (now the buffer is full); (6) sort on `RecordID2`; load records from S, one at a time, and perform the refinement step. Repeat from step (3) until the source (STT join in that case) is exhausted. Hence, this strategy, by ordering the pairs of records to be accessed, saves numerous page faults.

The resulting QEP includes two blocking iterators (*Build* and *SegSort*) between which the available buffer memory must be split. Basically there are two strategies for memory allocation for such QEPs:

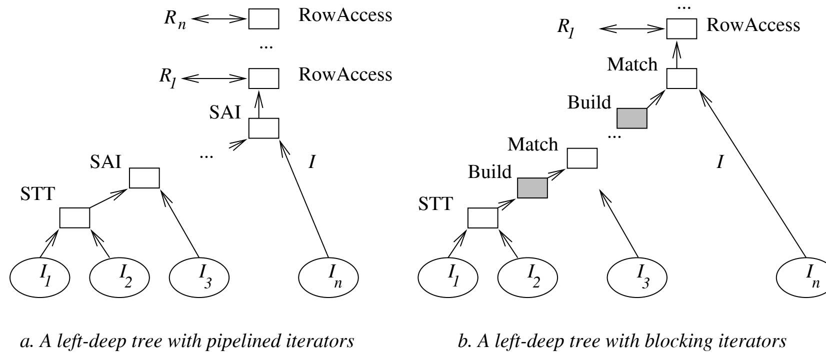
1. *Flush intermediate results*. This is the simplest solution: the total buffer space M allocated to the QEP is assigned to the *Build* iterator, the result of the join (STT) is flushed onto disk and the total space M is then reused for *SegSort*. The price to be paid is a possibly very large amount of write and read operations onto (from) disk for intermediate results.
2. *Split memory* among iterators and avoid intermediate materialization. Each of the iterators of the QEP is assigned part of the global memory space M. Then intermediate results are kept in memory as much as possible but less memory is available for each iterator [BKV98,ND98].

4 Spatial Join Query Processing

Using the above platform, our objective is to experimentally evaluate strategies for queries involving one or several spatial joins in sequence.

Fig. 3 illustrates two possible QEPS for processing query $R_1 \bowtie R_2 \ldots \bowtie R_n$, using index $I_1, I_2, \ldots I_n$., which both assume (i) the optimizer tries to use as much as possible existing spatial indices when generating QEPs and (ii) that the n-way join is first evaluated on the MBRs (filter step) and then on the exact geometry: an n-way join is performed on a limited number of tuples of the cartesian product $R_1 \times R_2 \times ...R_n$ (refinement step, requiring n row accesses). Both QEPS are left-deep trees [Gra93]. In such trees the right operand of a join is always an index, as well as the left operand for the left-most node. Another approach, not investigated here, would consists in an n-way STT, i.e., a synchronized traversal of n R-trees down to the leaves. See [MP99] for a comprehensive study.

The first strategy (Fig. 3.a) is fully pipelined: a STT join is performed as the left-most node, and a SAI join is executed for the following joins: at each step a new index entry `[MBR, RecordID]` is produced. The MBR is the argument of a window query for the following join. The result is a tuple $i_1, i_2, \ldots i_n$ of record id: the records are then retrieved with *RowAccess* iterators, one for each relation, in order to perform the refinement step on the n-way join but on a limited number of records. The second strategy (Fig. 3.b) uses instead of SAI the Build-and-Match strategy.



a. A left-deep tree with pipelined iterators *b. A left-deep tree with blocking iterators*

Fig. 3. Two basic strategies for left-deep QEPs

Evidently, the QEPs shown in Fig. 3 are extreme cases. Depending on the estimated size of output(s), a merge of both strategies can be used for a large number of joins. More importantly, the refinement step can be done prior to the completion of the query if it is expected that the candidate set contains a large number of false hits. By computing the refinement step in a lazy mode, as suggested in Fig. 3, the cardinality of intermediate results is larger (because of false hits) but the size of records is smaller.

We do not consider the case of bushy trees since they involve joins algorithms upon non-indexed relations. As an example of bushy-tree QEP, consider the following QEP for the join of 4 relations R_1, R_2, R_3, R_4: R_1 and R_2 are joined using STT as well as R_3 and R_4. The two (non-indexed) intermediate results must then be joined. In the case of n$\leqslant$3 (only one or two joins) which will be considered here, only left-deep trees can be generated.

Join strategies

We describe in this section the three variants of the same strategy called *Build-and-Match* (Fig. 3.b) which consists in building on the fly an index on a non indexed intermediate relation and to join the result with an indexed relation. When the structure built is an R-tree, then the construction is followed by a regular STT join. The rationale of such an approach is that even though building the structure is time consuming, the join behind is so efficient that the overall time performance is better than applying SAI. Of course the building phase is implemented by a blocking iterator and requires memory space.

STJ
The first one is the Seeded Tree Join (STJ) [LR98]. This technique consists in building from an existing R-tree, used as a *seed*, a second R-tree called *seeded R-tree*. The motivation behind this approach is that tree matching during the join phase should be more efficient than if a regular R-tree were constructed. During the *seeding phase*, the top k levels of the seed are copied to become the top k levels of the seeded tree. The entries of the lowest level are called *slots*.

During the *growing phase*, the objects of the non indexed source are inserted in one of the slots: a rectangle is inserted in the slot that contains it or needs the least enlargement. Whenever the buffer is full, all the slots which contain at least one full page are written in temporary files.

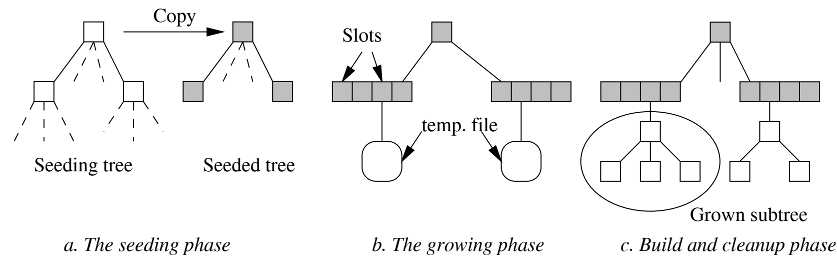


Fig. 4. Seeded tree construction

When the source has been exhausted, the construction of the tree begins: for each slot, the objects inserted in the associated temporary files (as well as the objects remaining in the buffer) are loaded to build an R-tree (called a *grown subtree*): the slot entry is then modified to point to the root of this grown subtree. Finally a *cleanup phase* adjusts the bounding boxes of the nodes (Fig. 4), as in classical R-trees.

The grown subtrees may have different heights: hence the seeded tree is not balanced. It can be seen as a forest of relatively small R-trees: one of the expected advantages of the method is that the construction of each grown subtree is done in memory.

There is however an important condition to fulfill: the buffer must be large enough to provide at least one page to each slot. If this is not the case, the pages associated to a slot will be read and written during the growing phase, thus rendering the method ineffective.

STR

The second Build-And-Match variant implemented, called Sort-Tile-Recursive (STR), constructs on the fly a STR packed R-tree [LEL97]. We also experimented the Hilbert packed R-tree [KF93], but found that the comparison function (based on the Hilbert values) was more expensive than the centroid comparison of STR.

The algorithm is as follows. First the rectangles from the source are sorted[3] by x-coordinate of their centroid. At the end of this step, the size N of the dataset is known: this allows to estimate the number of leaf pages as $P = \lceil N/c \rceil$ where c is the page capacity. The dataset is then partitioned into $\lceil \sqrt{P} \rceil$ vertical slices. The $\lceil \sqrt{P} \rceil .c$ rectangles of each slice are loaded, sorted by the y-coordinate

[3] The sort is implemented as an iterator which carries out a sort-merge algorithm according to the design presented in [Gra93].

of their center, grouped into runs of length c and packed into the R-tree leaves. The upper levels are then constructed according to the same algorithm. At each level, the nodes are roughly organized in horizontal or vertical slices (Fig .4).

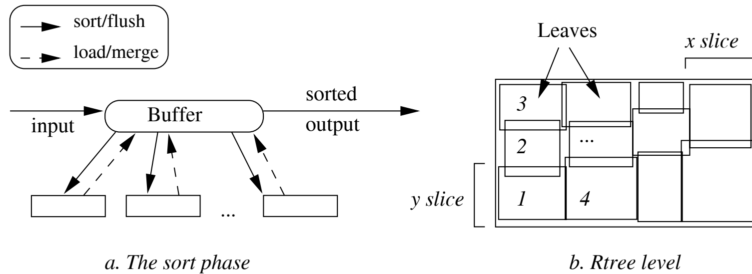


Fig. 5. STR tree construction

SaM

The third Build-And-Match variant called Sort-and-Match (SaM) is novel. It uses the STR algorithm but the construction is stopped at the leaf level, and the pages are not written onto disk. As soon as a leaf l has been produced, it is joined to the existing R-tree I_R: a window query with the bounding box of l is generated which retrieves all I_R leaves l' such that $l.MBR$ intersects $l'.MBR$. l and l' are then joined with the plane-sweep algorithm already used in the STT algorithm.

An interesting feature of this algorithm is that, unlike the previous ones, it does not require the entire structure to be built before the matching phase thus saving the flushing of this structure onto disk, resulting in much faster response time.

5 Performance Evaluation

The machine used throughout the experiments is a SUN SparcStation 5 with 32 MB of memory, running SunOS 4.1. We use in our experiments synthetic datasets, created with the ENST rectangle generator [GOP+98]. This tool[4] generates a set of rectangles according to a statistical model whose parameters (size, coverage, distribution) can be specified. The 3 following statistical models were used sharing the same 2D universe (map):

1. *Counties* (called *Biotopes* in [GOP+98]) simulates a map of counties; rectangles have a shape and location uniformly distributed, and the overlap (ratio between sum of the areas of the rectangles and map area) is 100%.

[4] Available at `http://www-inf.enst.fr/ bdtest/sigbench/`.

Nb records	Pages	Size (MB)
20K	769	3.1
40K	1539	6.3
60K	2308	9.4
80K	3078	12.6
100K	3846	15.7

Datafiles

Dataset	Pages	Levels	Root entries
COUN20	169	2	166
COUN40	330	3	3
COUN60	483	3	4
COUN80	650	3	6
COUN100	802	3	8

R*trees

Fig. 6. Database sample

2. *Cities* [GOP+98] simulates a map of cities: the map contains small rectangles whose shape is normally distributed (around the square shape) and whose location is uniformly distributed. The overlap is equal to 5%.
3. *Roads* simulates a map of roads: rectangles location and shape is uniform as in *Counties* but overlap is 200%.

For each of the statistical models, 5 datasets have been generated, with a size ranging from 20 000 to 100 000 objects referred to as DATxx, where DAT is in COUN, CIT, ROA and xx ranges from 20 to 100. For example, COUN20 stands for *Counties* with 20 000 rectangles.

Join strategies are evaluated on the query *Cities* $\bowtie$ *Counties* in the case of single joins and the query *Cities* $\bowtie$ *Counties* $\bowtie$ *Roads* for two-way joins.

We assume a page size of 4K and a buffer size ranging from 400K (100 pages) to 2.8MB (700 pages). The record size is 158 bytes and the buffer policy is LRU. Fig. 6 gives some statistics on the generated database (data file and index). Only the information on *Counties* is reported. Indeed the sizes do not depend on the statistical model, so *Cities* and *Roads* have almost identical characteristics. The fanout (maximum number of entries/page) of an R*tree node is 169. We give the number of entries in the root since it is an important parameter for the seeded tree construction.

The main performance criteria are (i) the number of I/O, i.e., the number of calls (page faults) to the I/O module and (ii) the CPU consumption.

The latter criteria depends on the algorithm. It is either measured as the number of comparisons (when sorting occurs), or the number of rectangle intersections (for join) or the number of unions (for R-tree construction): see Appendix A. The parameters chosen are the buffer size, the data set size and of course, the variants in the query execution plan and the join algorithms.

Single Join

When there is a single join in the query and both relations are indexed, a good candidate strategy is STT. Part of our work below is related to a closer assessment of this choice. To this end, we investigate the behavior of the candidate algorithms for single joins, namely SAI and STT. Fig. 7 gives for each algorithm, the number of I/Os as well as the number of rectangle intersection

tests (NBI), for a buffer set to 250 pages (1 MB). STT_{RA} stands for a QEP where the join is followed by a *RowAccess* operator, while STT is a stand alone join. Indeed, SAI and STT_{RA} deliver exactly the same result, namely pairs of `[Record, RecordID]`, while STT only yields pairs of `RecordID`.

As expected, the larger the dataset, the worse is SAI performance, both in I/Os and NBIs. There is a significant overhead as the R*tree size is larger than the available buffer. This is due to the repeated execution of window queries with randomly distributed window arguments.

STT outperforms SAI with respect to both I/Os and NBI. But as explained above, the comparison to be done is not between SAI and STT but between SAI and STT_{RA}. Then, looking at Fig. 7, the number of I/Os is of the same order for the two algorithms. Furthermore, it is striking that the *RowAccess* cost is more than one order of magnitude larger than the join itself for STT (e.g., for a dataset size of 100K, there are 104 011 I/Os while the join phase costs only 1 896 I/Os)!

The *RowAccess* iterator in the QEP implementing STT_{RA}, reads the pages at random. Then a large number of pages are read more than once. The number of I/Os depends both on the buffer size and on the record size (here 158, which is rather low) and can be estimated according to the model in [Yao77].

Since STT's performance (without *RowAccess*) is not very sensitive to an increase in the index size, it should not be very sensitive to a decrease in memory space. This justifies that most of the buffer space available should be dedicated to the *RowAccess* iterator in order to reduce its extremely large cost.

	Dataset size							
	40 000		60 000		80 000		100 000	
	I/Os	NBI	I/Os	NBI	I/Os	NBI	I/Os	NBI
SAI	19 761	11 741	49 885	20 399	81 206	27 032	114 805	33 855
STT_{RA}	37 557	2 576	59 826	4 000	81 165	5 431	104 011	6 898
STT	755	2 576	1 084	4 000	1 570	5 431	1 896	6 898
Result size:	116 267		171 343		228 332		288 846	

Fig. 7. Left-most node: join Cities-Counties, buffer size = 250 pages

To reduce the number of datafile accesses, we insert in the QEP a *SegSort* iterator before the *RowAccess*. Pages whose ids are loaded in the *SegSort* buffer can then be read in order rather than randomly. The efficiency depends on the size *SGB* of this buffer.

Fig. 8 displays the number of I/Os versus the data size, for SAI and STT, for several values of the parameter *SGB*. The total buffer size is 250 pages, and is split into a buffer dedicated to *SegSort* and a 'global' buffer whose size is 250 - *SGB*. STT-xx stands for the STT join where *SGB*=xx. In order to compare with the results of Fig. 7, we only access one relation. The larger *SGB*, the larger the gain. This is due to the robustness of STT performance with respect

to buffer size: its performance is not significantly reduced with a small dedicated buffer size.

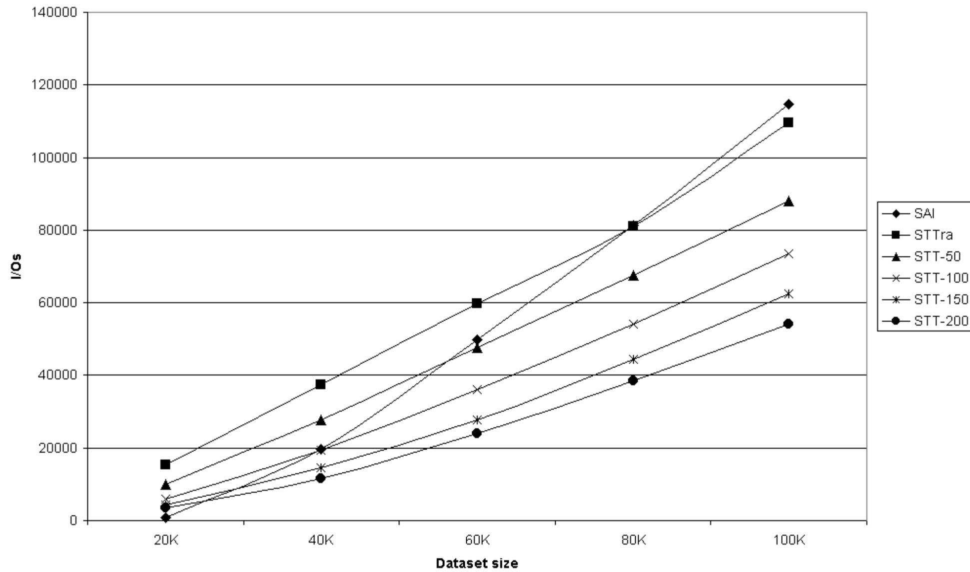


Fig. 8. SegSort experiment

Figure 8 illustrates the gain from sorting the pages to be accessed: for a large data set size, the gain with STT-200 is almost 3, compared to STT_{RA}.

In conclusion, the combination of STT with a *SegSort* operator (or any other mean to reduce the cost of random I/Os, for instance spatial data clustering) outperforms SAI.

We now compare the performance of the 3 Build-And-Match candidate algorithms (STJ, STR and SaM). Both the *Build* and the *Match* phases are considered, but we do not account for any *FileScan*. In other words, as stressed above, we restrict to the case where the join is executed on an intermediate result in which each tuple is produced one at a time.

Figure 9.a displays the cost of the 3 algorithms for 4 data set sizes. The case of STJ deserves some discussion. Note first that it is very unlikely that we can copy more than the root of the seeding tree because of the large fanout (169) of the R*tree. Indeed, in copying the first level also, the number of slots would largely exceed the buffer size.

In copying only the root, the number of slots may vary between 2 and 169. Actually, in our database the root is either almost full (dataset size 20K) or almost empty (dataset size $\geqslant$ 40K). See Figure 6.

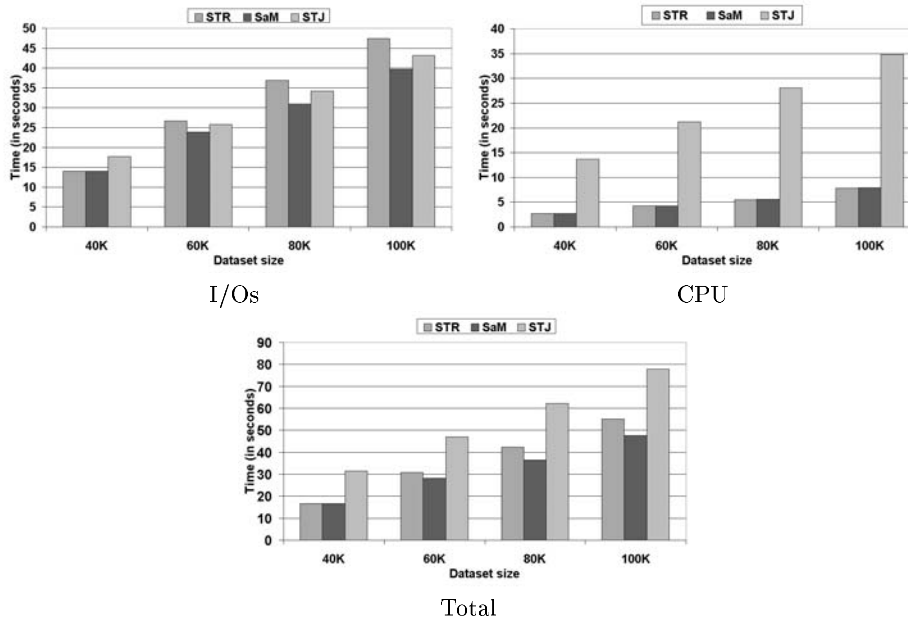


Fig. 9. Build and match joins

When the number of slots is large, one obtains a large number of grown R-trees (one per slot) whose size is *small*. Then the memory utilization is very low: an almost empty root with 2 or 3 leaves[5]. If the number of slots is small, then there is a small number of large R-trees, each of them requiring a significant construction time. In all cases, the CPU construction cost is high, although the I/Os cost is low because each grown subtree can be built in memory.

STR and SaM are extremely efficient with small dataset sizes (40K). Indeed the construction of the index is entirely done in main memory. Even for large data sets, SaM is very efficient. Compared to STR, the number of rectangle intersections is the same, but since the tree is not constructed the number of I/Os is smaller, the more the data set size increases (it is 20 % smaller than for STR with a dataset size greater than 80K).

During the match phase, SaM is also efficient: in fact it can be seen as a sequence of window queries, with two major improvements: (i) leaves are joined, and not entries, hence one level is saved during tree traversal, and (ii) more importantly, two successive leaves are located in the same part of the search space. Therefore the path in the R-tree is likely to be already loaded in the buffer.

[5] We do not pack the roots of grown subtrees, as proposed in [LR98]. This renders the data structure and implementation complex, and has some further impact on the design of the STT.

We now test the robustness of the algorithms performance with respect to the buffer size. In Figure 10, we measure the performance of the algorithms by joining two 100K datasets and letting the buffer size vary from 100 pages (400K) to 700 pages (2.8 MB). *RowAccess* is not taken into account. We do not include the cost of STJ for the smallest buffer size since buffer thrashing cannot be avoided in that case.

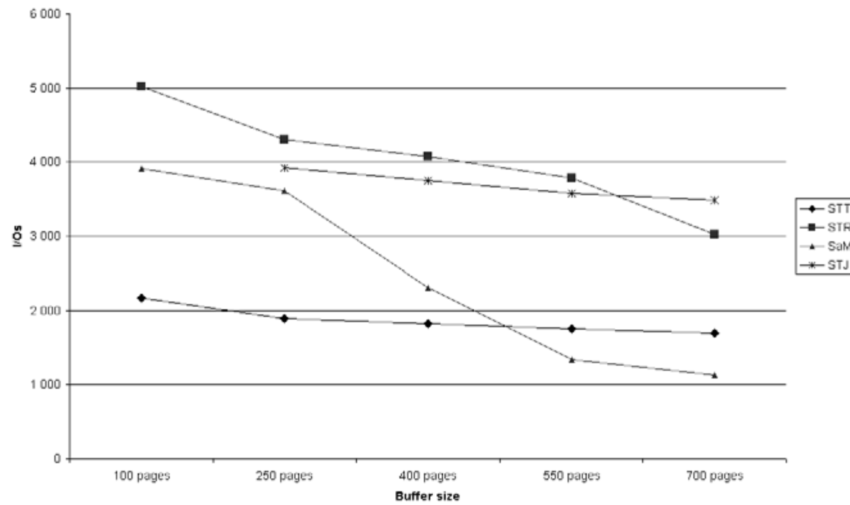


Fig. 10. JOIN Cities 100K - Counties 100K, varying buffer

Looking at Figure 10, the following remarks are noteworthy: (i) the sort-based algorithms benefit from large buffers; this is less clear for STJ; (ii) as expected, STT performance is robust with respect to buffer size; this is important since algorithms whose memory requirement is known and reasonable in size allow for more flexibility when assigning memory among several operators, as shown in the next section. Observe also that when the *Build* phase can be performed in memory, the *Join* phase of SaM outperforms STT; (iii) the larger the buffer size, the more SaM outperforms the two other Build-And-Match strategies: while its gain over STR is only 20% for small buffer size, it reaches three for a buffer capacity of 700 pages.

Two way joins

This section relies on the above results for the evaluation of QEPs involving two joins. In the sequel, the left-most node of the QEP is always an STT algorithm performed on the two existing R*trees (on *Cities* and *Counties*) which delivers pairs of index entries $[i_1, i_2]$. The name of a join algorithm denotes the

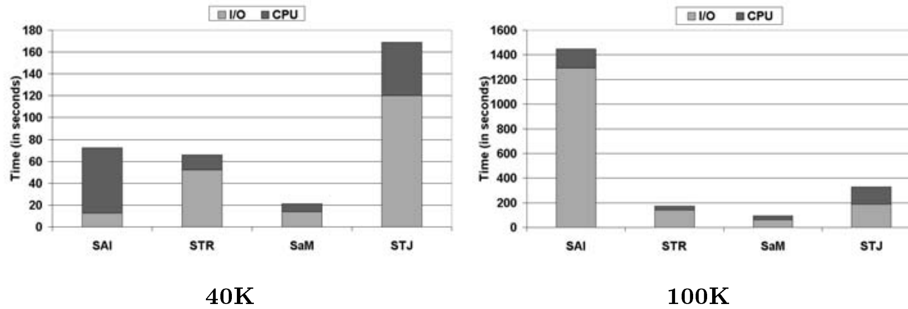


Fig. 11. Two way joins

second join algorithm, which takes the result of STT, builds a structure and performs the join with the index on *Roads*.

Note that in that case one does not save a *RowAccess* with SAI for the refinement step. Indeed as the Build-And-Match strategies, SAI reads as an entry only an index entry `[RecordID,MBR]` from the STT join. The result is, in all cases, a set of triplets of index entries.

The datasets *Counties*, *Cities* and *Roads*, have equal size, and a fixed buffer of 500 pages has been chosen. We make the experiments for the medium size of 40K and the larger size of 100K. The latter 2 way-join yields 865 473 records, while the former 314 617 records.

Figure 11 gives the response time for SAI and the three variants of Build-And-Match algorithms. Let us look first at SAI performance. For a small dataset size (40K), the index fits in memory, and only few I/Os are generated by the algorithm. However the CPU cost is high because of the large number of intersection tests. For large dataset sizes, the number of I/Os is huge, rendering this algorithm definitely not the right candidate.

STJ outperforms SAI for large datasets. But its performance is always much below that of SaM and STR. The explanation of this discrepancy is the following. For a 40K size, the first level of the seeding tree could be copied, resulting into 370 slots. The intermediate result consists of 116 267 entries. So, there is an average of 314 entries per slot: each subtree includes a root with an average of two leaves, leading to a very bad space utilization. A large number of window queries are generated due to the unbalance of the matched R-tree. In the case of 100K datasets, only 8 slots can be used, and the intermediate result consists of 288 846 records. Hence we must construct a few, large R-trees, which is very time consuming.

SaM significantly outperforms STR, mostly because it saves the construction of the R-tree structure, and also because the join phase is very efficient. It is worth noting, finally, that SAI is a good candidate for small datasets sizes, although its CPU cost is still larger. One should not forget that SAI is, in that case, the only fully pipelined QEP. Therefore the response time is very short, a

parameter which can be essential when the regularity of the data output is more important than the overall resource consumption.

Discussion

By considering complete QEPs, including the I/O operations for the refinement step, we were able to identify the bottlenecks and the interactions between the successive parts of a QEP.

The efficiency of the commonly accepted STT algorithm is natural: an index is a small, structured collection of data, so joining two indices is more efficient than other strategies involving the data files. The counterpart, however, is the cost of accessing the records after the join for the refinement step, whose cost is often ignored in evaluations, although several papers report the problem (see for instance [PD96] and the recent work of [AGPZ99]). It should be noted that in pure relational optimization, the manipulation of `RecordID` lists has been considered for a long time to be less efficient than the (indexed) nested loop join [BE77]. Even nowadays, the ORACLE DBMS does use a SAI strategy in the presence of two indices [Ora]. In the context of spatial databases, though, SAI provides a prohibitive cost as soon as the index size is larger than the buffer and the number of window queries is high. Whenever STT is chosen, we face the cost of accessing the two relations for the refinement step. When data is not spatially clustered, the present experiment suggests to introduce a scheduling of row accesses through a specific iterator. We used the algorithm of [Val87], but other techniques are available [Gra93]. The combination of STT and *SegSort* outperforms SAI for large datasets, in part because of the robustness of STT with respect to the buffer size.

For two-way joins, the same guidelines should apply. Whenever we intend to build an index for subsequent matching with an existing R-tree, the build algorithm performance should not degrade when there is a shortage of buffer space, since most of the available space should be dedicated to the costly access to records after the join. We experimented three such Build-And-Match strategies: a top-down index construction (STJ), a bottom-up index construction (STR) and an intermediate strategy which avoids the full index construction (SaM). Several problems were encountered with STJ, while the classical solutions based on sorting appear quite effective. They provide a simple, robust and efficient solution to the problem of organizing an intermediate result prior to its matching with an existing index. The SaM algorithm was shown to be a very good candidate: it can be carried out with reasonably low memory space and provides the best response time since its *Build* phase is not completely blocking: records are produced before the build phase is completed.

6 Conclusion and Future Work

The contribution of this paper is three fold: (i) provide an evaluation platform general enough to experimentally evaluate complex plans for processing spatial queries and to study the impact on performance of design parameters such as buffer size, (ii) show that in build-and-match strategies for spatial joins it was

not necessary to completely build the index before the join: this resulted into a join strategy called SaM that was shown in our experiment to outperform the other known build-and-match strategies, (iii) show that physical operations that occur in the query execution plan associated with a join strategy have a large impact on performance. For example, we studied the impact of record access after the join, which is a very costly operation.

The performance evaluation stressed the importance of memory allocation in the optimization of complex QEPs. The allocation of available buffer space among the (blocking) operators of a QEP, although it has been addressed at length in a pure relational setting, it is still an open problem. We intend to refine our evaluation by studying the impact of selectivity and relation size on the memory allocation. Some other parameters such as the data set distribution or the placement of the record access in the QEP may also have some impact. The aim is to exhibit a cost model simple enough to be used in an optimization phase to decide for memory allocation.

References

[AGPZ99] D. Abel, V. Gaede, R. Power, and X. Zhou. Caching Strategies for Spatial Joins. *GeoInformatica*, 1999. To appear.

[APR+98] L. Arge, O. Procopiuc, S. Ramaswami, T. Suel, and J. Vitter. Scalable Sweeping Based Spatial Join. In *Proc. Intl. Conf. on Very Large Data Bases*, 1998.

[BE77] M. Blasgen and K. Eswaran. Storage and access in relational databases. *IBM System Journal*, 1977.

[BKK96] S. Berchtold, D. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *Proc. Intl. Conf. on Very Large Data Bases*, 1996.

[BKS93] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1993.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*tree : An Efficient and Robust Access Method for Points and Rectangles. In *Proc. ACM SIGMOD Intl. Symp. on the Management of Data*, pages 322–331, 1990.

[BKSS94] T. Brinkhoff, H.P. Kriegel, R. Schneider, and B. Seeger. Multi-Step Processing of Spatial Joins. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 197–208, 1994.

[BKV98] L. Bouganim, O. Kapitskaia, and P. Valduriez. Memory Adaptative Scheduling for Large Query Execution. In *Proc. Intl. Conf. on Information and Knowledge Management*, 1998.

[GOP+98] O. Gunther, V. Oria, P. Picouet, J.-M. Saglio, and M. Scholl. Benchmarking Spatial Joins *À La Carte*. In *Proc. Intl. Conf. on Scientific and Statistical Databases*, 1998.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GS87] R.H. Güting and W. Schilling. A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem. *Information Sciences*, 42:95–112, 1987.

[Gun93] O. Gunther. Efficient Computation of Spatial Joins. In *Proc. IEEE Intl. Conf. on Data Engineering*, pages 50–59, 1993.

[Gut84] A. Guttman. R-trees : A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD Intl. Symp. on the Management of Data*, pages 45–57, 1984.

[HJR97] Y.-W. Huang, N. Jing, and E.A. Rudensteiner. Spatial Joins Using R-trees: Breadth-first Traversal with Global Optimizations. In *Proc. Intl. Conf. on Very Large Data Bases*, 1997.

[KF93] I. Kamel and C. Faloutsos. On Packing Rtrees. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, 1993.

[KS97] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1997.

[LEL97] S. Leutenegger, J. Edgington, and M. Lopez. STR: a Simple and Efficient Algorithm for Rtree Packing. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, 1997.

[LR96] M.-L. Lo and C.V. Ravishankar. Spatial Hash-Joins. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 247–258, 1996.

[LR98] M.-L. Lo and C.V. Ravishankar. The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins. *IEEE Transactions on Knowledge and Data Engineering*, 10(1), 1998. First published in SIGMOD'94.

[MP99] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for joining multiple inputs. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1999.

[ND98] B. Nag and D. J. DeWitt. Memory Allocation Strategies for Complex Decision Support Queries. In *Proc. Intl. Conf. on Information and Knowledge Management*, 1998.

[NHS84] J. Nievergelt, H. Hinterger, and K.C. Sevcik. The Grid File: An Adaptable Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[Ora] Oracle 8 Server Concepts, Chap. 19 (The Optimizer). Oracle Technical Documentation.

[Ore86] J. A. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 326–336, 1986.

[PD96] J.M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 259–270, 1996.

[RL85] N. Roussopoulos and D. Leifker. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 17–26, 1985.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 507–518, 1987.

[Val87] P. Valduriez. Join Indices. *ACM Trans. on Database Systems*, 12(2):218–246, 1987.

[VG98] V.Gaede and O. Guenther. Multidimensional Access Methods. *ACM Computing Surveys*, 1998. available at http://www.icsi.berkeley.edu/ oliverg/survey.ps.Z.

[Yao77] S. B. Yao. Approximating Block Accesses in Data Base Organizations. *Communication of the ACM*, 20(4), 1977.

Appendix A

We give below a simple cost model for estimating the response time of an algorithm (query), which includes both I/Os and CPU time. For the I/O time calculation, we just assume that each I/O, i.e., that each disk access has a fixed cost of 10msec. Therefore, if nb_io denotes the number of I/Os, the time cost (in seconds) due to the disk is:

$$T_{disk} = nb_io \cdot 0.01 \quad (1)$$

In order to estimate CPU time, we restricted to the following operations: rectangle intersections, rectangle unions and sort comparisons. The parameters are then: (a) the number of rectangle intersections nb_inter, (b) the number of number comparisons nb_comp and (c) the number of rectangle unions nb_union. Since we consider a two-dimensional address space (generalizations are straightforward), each test for rectangle intersection costs four CPU instructions (two comparisons per dimension). Also, each rectangle union costs four CPU instructions. Finally, each comparison between two numbers costs one CPU instruction. If MIPS denotes the number of instructions executed in the CPU per second, then the time for each operation is calculated as:

$$T_{inter} = \frac{nb_inter \cdot 4}{MIPS} \cdot 10^{-6} \quad (2)$$

$$T_{union} = \frac{nb_union \cdot 4}{MIPS} \cdot 10^{-6} \quad (3)$$

$$T_{comp} = \frac{nb_comp}{MIPS} \cdot 10^{-6} \quad (4)$$

The CPU cost is thus estimated as

$$T_{proc} = T_{inter} + T_{union} + T_{comp} \quad (5)$$

In addition to the above CPU costs, we assume that each read or write operation contributes to a CPU overhead of 5000 CPU instructions for pre and post processing of the page:

$$T_{prep} = \frac{nb_io \cdot 5000}{MIPS} \cdot 10^{-6} \quad (6)$$

The total CPU cost is then

$$T_{CPU} = T_{prep} + T_{proc} \quad (7)$$

The response time of a query is then estimated as:

$$T_{response} = T_{CPU} + T_{disk} \quad (8)$$