# The BASIS System: a Benchmarking Approach for Spatial Index Structures*

C. Gurret[2], Y. Manolopoulos[1], A.N. Papadopoulos[1] P. Rigaux[2]

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

**Abstract.** This paper describes the design of the BASIS prototype system, which is currently under implementation. BASIS stands for **B**enchmarking **A**pproach for **S**patial **I**ndex **S**tructures. It is a prototype system aiming at performance evaluation of spatial access methods and query processing strategies, under different data sets, various query types, and different workloads. BASIS is based on a modular architecture, composed of a simple storage manager, a query processor, and a set of algorithmic techniques to facilitate benchmarking. The main objective of BASIS is twofold: (i) to provide a benchmarking environment for spatial access methods and related query evaluation techniques, and (ii) to allow comparative studies of spatial access methods in different cases but under a common framework. We currently extend it to support the fundamental features of spatiotemporal data management and access methods.

**Keywords:** benchmarking, spatial access methods, spatiotemporal data management, performance evaluation

## 1 Introduction

Spatial data management has been an active area of research over the past ten years [LT92,G94]. A major component of such research efforts has been, and continues to be, concerned with the design of efficient spatial data indexing methods which reduce query response time during spatial query processing [Gut84,SRF87,HSW89,?,KF94]. Some promising proposals have been reported in the literature aimed at enhancing these indices with additional, specific query handling capabilities [FBF77,Are93,BKS93,RKV95]; a spatial data index may be specifically redesigned to support region queries, nearest-neighbor queries or spatial join queries. Another direction is the establishment of robust cost models for specific indices and query processing algorithms, i.e functions which estimate the cost of processing spatial queries. As spatial data management develops, particularly in spatiotemporal applications, we expect further proposals for indexing methods to appear.

* 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.

The various index methods, then, behave differently according to certain settings, such as the buffering strategies and query profiles. Therefore, for any comparative performance test, the tester must be careful when declaring a clear winner. Moreover, all these methods have been proposed independently and, consequently, there is a lack of a general framework under which an extensive comparative study can be performed. Although several comparative studies have been performed, there is a lack of a spatial benchmark that could be used as a yardstick towards spatial indexing method design, implementation and performance evaluation.

As a solution to this problem, we are constructing an integrated benchmark development and execution environment: **B**enchmarking **A**pproach for **S**patial **I**ndex **S**tructures (BASIS). This is a platform which facilitates the incremental integration of multiple indexing methods, query algorithms, and data sets for use in benchmark runs. So far, the type of benchmarks supported are those designed to test only indexing methods and query processing algorithms.

Related work can be divided in three categories: (i) benchmarking for DBMS, (ii) generic index support environments, and (iii) recent benchmarking platforms. The first one has the longer history. To test relative performance of different DBMS, the research and industry community has sought a standard benchmark for all systems. Early benchmarks for DBMS included DebitCredit [Anon et al.85] and Wisconsin [BDT83] which measured the number of transactions per second (TPS) and produced price/performance ratings for systems in \$/TPS. Later, the Transaction Processing Performance Council (TPC) published a number of benchmarks for specific application domains such as OLAP [TPCC97] and decision support systems [TPCD95]. With the emergence of object data bases a number of new benchmarks appeared; see [Cha95] for an annotated bibliography. This type of benchmark differs from the type supported by BASIS in that they attempt to test entire systems. More specifically, BASIS supports researchers working on particular modules of a complete system: index and query processor.

Other related systems include the DEVise environment [LRB$^+$97]. This is a visualization tool for index structures. The GiST project [HNP95] has produced a generic software package which can be used to generate different tree index structures based on the B-tree paradigm.

Systems with a similar scope to that of BASIS include A La Carte [GOP$^+$98] and TimeIT [KS95]. The former is a benchmarking environment for testing spatial query processing algorithms. BASIS generalizes this objective by allowing the testing of index structures as well. TimeIT is a benchmarking environment for testing algorithms for temporal query processing.

The rest of the paper is organized as follows. The next section lists and discusses requirements for a benchmarking environment. Section 3 illustrates the architecture of BASIS. Section 4 discusses issues for spatial and spatiotemporal data management. An example benchmark is explained in Section 5. Finally, Section 6 concludes the paper.

## 2 Requirements for a Benchmarking Environment

The general requirement for a benchmarking environment is to aid a user to set up, run, and analyze the results of a benchmark. This process is decomposed into the following more specific requirements:

1. provide a rich variety of data sets of different data object types,
2. integrate different index structures,
3. integrate different query processing algorithms,
4. facilitate the analysis of benchmark results,
5. provide visualization methods for viewing data sets, index structures, and benchmark results

In the sequel, we discuss each topic in detail.

### 2.1 Datasets

The first fundamental requirement for a database benchmark is data. A benchmarking environment must provide the benchmark designer with as much data from the appropriate application domain as he may need. The nature of the data used in a benchmark run can greatly influence results. In the spatial application domain, some index structures are specifically designed for point data. Other index structures outperform the previous set of indexes when the data represent objects with extension, such as lines and regions. The benchmarking environment must therefore provide a variety of data to accommodate this aspect of benchmarking. At least part of the data must be real-life data. However, with the current poverty of large scale public real-life data, artificial data may also be provided.

### 2.2 Index structures

The environment should contain a number of different index structures. They might be provided as part of the platform (for instance, we already implemented several variants of the R-trees) or be added by some user to perform new experiments and comparisons. This involves some design issues. First, one must be able to use an existing structure without requiring a detailed knowledge of its internal implementation. ¿From a practical point of view, this means that all structures should share a common design for the essential operations: data loading, scan, point and window queries, etc.

Second, at a lower level, the platform must provide the building blocks to create a new structure with minimal effort. This is necessary because, one the one hand, we want to save the tedious task of programming the memory management and I/O functionalities, and because, on the other hand, a strong integration of the new structure in the platform implies first that the design is made according to some generic “pattern”, and second that the implementation takes advantage as much as possible of low-level existing functionalities.

## 2.3 Query processing

As with indices, the environment should contain a number of different query processing algorithms. Again, these should accumulate within the environment without penalty to the implementation effort. Moreover, the techniques and algorithms already integrated should benefit to any researcher who wants to extend the platform.

A significant ambition of the current work is to provide an open query processor which proposes a set of commonly used operators (such as external sorting for instance) and simultaneously allows to integrate the existing operators with new algorithms. We consider this requirement as very important because it does not limit the benchmarking work to the comparison of some isolated structures or algorithms, but permits to build arbitrarily complex algorithms by "branching" together some operators extracted from the available library.

We illustrate this approach with two examples. Imagine first that we want to compare to algorithms, $A_1$ and $A_2$, but the result of $A_1$ consists of records sorted on some attribute, while the result of $A_2$ consists of record ids (i.e, the address of each record on the disk). Clearly, a performance comparison of both algorithms is not fair: we must complete $A_2$ with two operators, namely a first one which fetches the records on the disk, and a second one which sorts the set of records.

More generally, in a context of spatiotemporal databases, one must be ready to mix several structures and algorithms on spatial, temporal and spatiotemporal data. The design of a benchmarking environment should allow such a merge of techniques issued from different fields, hopefully in a clean and elegant setting which permits an accurate reporting of results. We propose in BASIS a query execution model which is general enough to encompass relational and spatial data and makes easy the integration of new methods.

## 2.4 Analysis of benchmark results

A benchmark may be intended to show the consumption of various system resources, such as disk accesses, as a function of some variable, e.g. data set size. The environment should allow a benchmark to produce statistics covering various variables in the benchmark and a variety of resources. It may also provide statistics on the performance of individual index functions, such as building, update, insertion, and search times.

The conventional representation of benchmark results is to plot a graph on the use of one resource against some varying parameter. Typically, such a graph shows that one index progressively outperforms the other as the size of the data set increases. Thus the investigator can justify claims about such relative performance. However, it is becoming increasingly important to be able to explain the reasons of differing performance. For instance, which factors cause the inferior index to eventually require almost double the number of disk accesses?

The authors in [KS97] considered this question. Their response is an example of the form of results analysis that BASIS is designed to support. They compared the performance of two spatial index structures, R*-tree and SR-tree, with a large number of nearest-neighbor queries. The main difference between the two methods is that R*-tree groups objects into minimum bounding rectangles while SR-tree groups them into minimum bounding circles. They found that SR-trees outperform the former structure. The authors then investigated this difference by re-analyzing the benchmark data and calculating the average volume and diameter of the bounding shapes in the leaf nodes of the structure. Here the diameter of a rectangle is its diagonal. This produced two further graphs of volume against data size and diameter against data size.

These graphs showed that the average volume of the rectangles was smaller than the circles but that the average diameter of the rectangles was larger. Put another way, R*-tree groups the points into small volume regions while SR-tree groups them into small diameter regions. All the queries of the benchmark were nearest-neighbor queries. Since short diameter is more significant for such queries than small volume, SR-tree is superior.

This work can be considered to be an ad hoc data mining process on benchmark results. A benchmarking environment should support such analysis as well as the simple conventional performance graphs.

### 2.5 Visualization

Visualization is a requirement in three areas: data sets, index structures, and results data. A user chooses data sets for a benchmark run that are appropriate for that benchmark. In particular, if artificial data are used, it is very difficult to verify that these data are appropriate. Some tool to give a visual representation of the space distribution of the data can aid the user greatly. Also, it can be of benefit to be able to view the structure of an index, for example tree fanout or height, when analyzing results. Finally, clear visual presentation of complex results is a key factor in their understanding and acceptance.

This is the set of requirements which have influenced the design of the BASIS system. So far the emphasis has been targeted to the first four requirements which are considered fundamental. The fulfillment of the last one is considered as future work. In the next sections we explain the way our system attempts to satisfy the fundamental requirements by illustrating the architecture and describing the internal components.

## 3 The BASIS Architecture

This section describes the fundamental components of the BASIS architecture. An outline of the architecture is depicted in Fig. 1. The platform has been implemented in C++ and runs on top of UNIX or Windows NT.

The platform is organized in three modules (Fig. 1), namely (1) the *storage manager* provides I/Os and caching services, (2) the *SAM toolkit* is a set of commonly used SAMs (Spatial Access Methods) and defines some design patterns which support an easy development of new structures, and finally (3) the *query* *processor* is a library of algorithms whose design follows the general framework of the *iterator model*, to be described below.

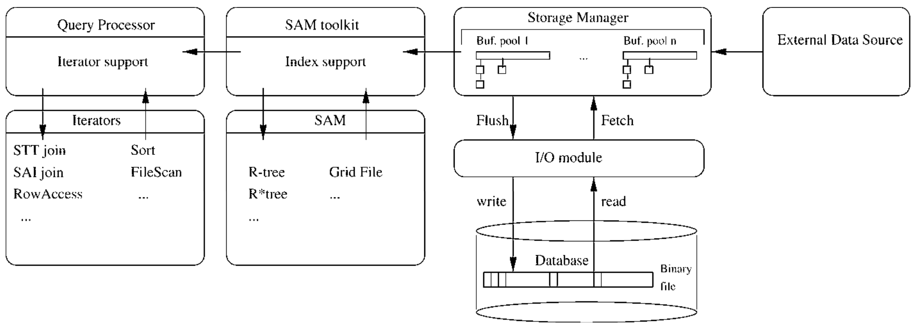


**Fig. 1.** The BASIS architecture

Each module defines a level in the architecture: this design allows an easy customization and extension. Depending on the query processing experiment, each level is easily extendible: the designer may add a new SAM, add a new spatial operator or algorithm at the query processor level, or decide to implement its own query processing module on top of the buffer management (I/O) module.

## 3.1 The storage manager

The storage manager is essentially in charge of managing a database. A database is a set of binary files which store either datasets (i.e., sequential collection of records) or SAMs. A SAM or index refers to records in an indexed data file through *record identifiers.*

Any binary file is divided into *pages* with size chosen at database creation. A page can be viewed (and accessed) as an array of bytes, but we provide a more structured and easy-to-use page representation. Under this second point of view, a page is structured as a *header* followed by an array of fixed-size *records.* Therefore, a page is formatted and this allows for an easy access to any record by its rank. Therefore, the interface to a formatted page is safer because invalid operations (such as an access beyond the limits of the pages) can be prevented.

Any information to be put in page is a `CStorable` object. The `CStorable` class defines the minimal behavior expected from such objects: essentially they must be able to dump/load their content to/from a given location. The storage manager provides a predefined list of such storables, including the atomic types `CString`, `CInt`, `CDouble`, the class `CRecordRef` which holds the address of a record, and the `CRect` class representing a bounding-box. By sub-typing the class `CStorable`, one can easily extend the storable types.

A *record* can now be built as a tuple of `CStorable` objects. Records are the main objects that BASIS deals with: a query execution plan, as described above, can essentially be seen as a tree of operators which consume and produce records. BASIS also provides a dynamic type management: any `CRecord` object knows its type and delivers this type as a `CType` object on demand. Conversely, any `CType` object can dynamically instantiate records.

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.

All algorithms requiring page accesses uniformly access these pages through the interface provided by the buffer manager.

## 3.2 The SAM toolkit

Here are now the two main types of files which are handled by the storage manager:

1. *Data files* are sequential collections of formatted pages storing *records* of a same type. Records in a data file can either be accessed sequentially (with a `FileScan` iterator, described below), or by their address (`RowAccess` iterator in the sequel).
2. *SAMs* are structured collections of `IndexEntry`. An index entry is a built-in record type with two attributes: the key and a record address. The key is the geometric key, usually the MBR.
   The currently implemented SAMs are a grid file, an R-tree, an R*-tree and several packed R-trees. Fig. 2 shows some of these variants, built on the hydrography dataset of the Connecticut State (Tiger Data)

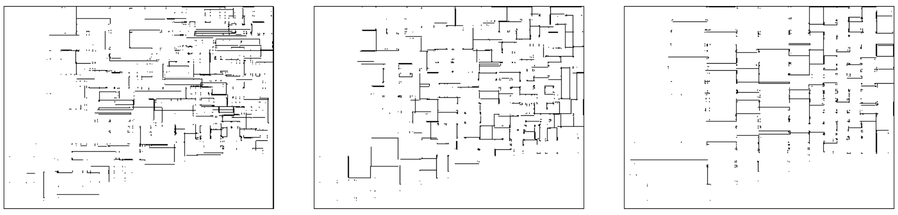


**Fig. 2.** R-tree variants (MBRs of leaf nodes) in BASIS

## 3.3 The query processor

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.

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.

Each iterator comes with three functions: `open`, `next` and `close`: `open` prepares each input source and allocates the necessary system resources, `next` runs one iteration step of the algorithm and produces a new record and `close` ends up the iteration and relaxes resources. This design offers two major advantages:

1. First it allows for simple QEP creation by "assembling" iterators together.
2. Second it is fairly easy to extend BASIS by adding a new iterator: providing that it defines the convenient interface as `open`, `next` and `close` (we provide the necessary C++ support for that), its integration in the BASIS iterator library is trivial.

We illustrate this feature with two examples of QEP on a same spatial join query $R \bowtie S$ (Fig. 3). We assume the existence of a spatial index $I_S$.

First (left of the figure), the QEP follows a the simple scan-and-index (SAI) strategy: relation $R$ is scanned with a `FileScan` iterator, and for each tuple $r$ in $R$, the `SAIjoin` iterator executes a window query on index $I_S$ with key $r.MBR$. This gives a record ID *RecordID2*. Finally the record with id *RecordID2* in the datafile $S$ is accessed with the `RowAccess` iterator.

Observe that BASIS dynamically constructs the type of the result: knowing the types of the records in the datafile $R$ and the index $I_S$, respectively, a simple bottom-up type inference allows to know the type of the records manipulated by each iterator, including the intermediate ones. This automatic type management is an important feature which facilitates the specification of QEPs.

Now, assume that a new join algorithm is proposed, which constructs some structure and matches it with the existing index. Then it suffices to reuse all the iterators, save the `SAIjoin` one, which is to be replaced by `NEWJoin` (Fig. 3, right part).

In both cases, the query is executed by using the sequence `open`, {`next`}, `close` on the root of the QEP (iterator `RowAccess`), where `next` denotes the retrieval of one record, involving a propagation of the operations down the tree. BASIS allows to collect the statistics on page faults and various other performance parameters during execution.

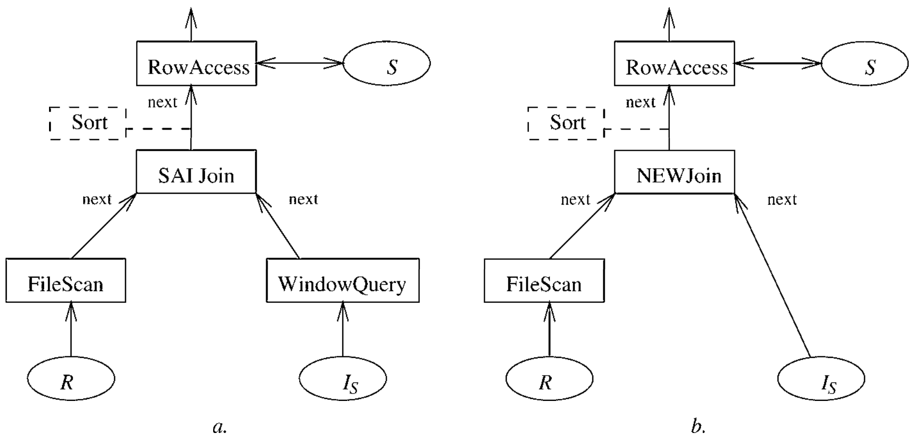


**Fig. 3.** Query execution plans

The platform also provides some basic modules such as sorting: the `Sort` iterator takes any flow of record, a comparison function, and performs an external sort/merge. As an example, one can introduce a `Sort` iterator[1] in the QEP of Fig. 3 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.

In summary, this query execution model permits the construction of an extendible library which defines a physical algebra for query evaluation. The modular design and the support for type inference and iterator development allows for an easy integration of new components and provides a convenient framework for new experiments. We currently enrich the platform with temporal (cooperation with the MUST group [DFS98]) and spatiotemporal structures and algorithms.

## 4 Spatial and Spatiotemporal Data Handling

Access method performance varies considerably by modifying the characteristics of the data. For example, highly clustered data usually degrades performance as opposed to uniformly distributed data. In the BASIS project we support two fundamental types of data:

- real-life data (e.g. Sequoia 2000 [SFGM93], TIGER/Line [Tiger94], etc),
- synthetic data, obeying various distributions (e.g. uniform, normal, exponential, etc).

[1] This iterator is *blocking*: almost all the job is done during the `open` operation. See Appendix B for a short description.

From one point of view, real-life data captures the characteristics of the data that real-life applications manipulate. However, to be able to change the data properties (population, distribution, coverage, etc.) synthetic data must be supported also. Table 1 summarizes the most important properties of synthetic data.

| **Property** | **Description** |
|---|---|
| data type | points, rectangles, line segments, polygons |
| population | how many objects are generated |
| distribution | uniform, normal, exponential, fractal |
| dimensionality | number of dimensions of address space (2-d, n-d) |
| coverage | how many objects cover a point of the address space |

**Table 1.** The most important properties of a synthetic data set.

Generally speaking, it is required to make the generated data seem less artificial than a simple uniformly distributed random placement of rectangles. Statistical properties of the generated data should be under the control of the user. In particular the user should exercise some control over the distribution of the objects in space and the size and shape of the objects. The data generator described in [GOP$^+$98] meets these requirements. The data are generated by controlling the fundamental parameters, and a large number of different data sets can be generated. These data are ported into BASIS internal representation to be part of the database. Real-life datasets can be used in the same way. This enables the generation of benchmarks based on real-life and synthetic but realistic data sets. If the user is not satisfied with the available data, new data sets can be generated and ported into BASIS. The performance evaluation reported in [GOP$^+$98] and [PRS99] is based on data sets generated by using the generator of ENST.

Spatial or spatiotemporal index structures can be build to support the underlying data sets. Currently, BASIS supports Grid-Files, R-trees and R*-trees in 1-D, 2-D or $N$-D spaces. Supporting multidimensional spaces is considered very important, since some applications require data manipulation in more than 2 or 3 dimensions. For example, some proposed spatiotemporal index mechanisms can be evaluated. The exploitation of 3-D R-trees is a straightforward solution to index multimedia objects, where the third dimension is time. BASIS supports such an index directly, by instantiating an R-tree or R*-tree in 3-D and passing as an input parameter the filename which contains the spatiotemporal data ($x$, $y$, $time$). Alternatively, a 2-D R-tree can be constructed to index the spatial attributes of the data and a 1-D R-tree can be utilized to index time intervals. Moreover, overlapping R-trees and RT-trees [XHL90] can be implemented very easily by performing modifications to the R-tree implementation.

Some primitive data types have been defined to enable indexing and query processing (see Fig. 4). The fundamental data types currently supported are *points*, *rectangles*, *line segments* and *point trajectories*. The point trajectory can

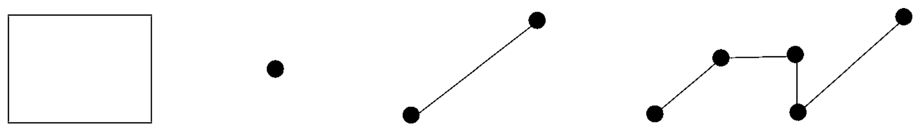


**Fig. 4.** Primitive data types supported.

be viewed as a collection of consecutive positions in the space. The main purpose of this type is to support spatiotemporal benchmarking for moving point databases. For example, a database of moving points can be generated by instantiating a number of point trajectories. Each trajectory corresponds to a collection of records. The user can use these predefined types to generate new ones. For example, rotated MBRs can be generated by inheriting from the `CRect` class and adding some new member variables (e.g. rotational angle). During instantiation of an object the dimensionality must be specified. The corresponding class definitions are implemented as templates, enabling an arbitrary number of dimensions with strict type checking.

Each spatial or spatiotemporal data type integrates some useful methods that enable processing. For example, the *intersects* method can be invoked between two rectangles or point trajectories to determine if they intersect or not; *inclusion* can be invoked to test if a point is enclosed by a rectangle. Since it is impossible to include every method that one may need, new methods can be defined using appropriate inheritance from the corresponding base classes.

So far the majority of benchmarks performed by several research efforts focus on spatial data only. We are currently working on performance evaluation of spatiotemporal access methods, and more specifically we focus on the case of moving point databases. In a moving point database each object is characterized by a point trajectory. The challenge is to index these trajectories to efficiently support *spatial-only*, *time-only*, and *spatiotemporal queries*. The first issue we raise is if the index should be *sparse* or *dense*. If a separate index entry is created for each object position then a dense index is created. On the other hand, if several consecutive trajectory positions are grouped together a sparse index is created. Evidently, a dense index occupies more space than a sparse index which is very critical when the database need to be updated frequently, and the number of moving objects is relatively large. On the other hand, the fact that a dense index holds every position of the trajectory, eliminates the need to search the trajectory itself during query processing, because all information is maintained in the index. It is not evident which strategy is best suited for the three aforementioned query types.

The second interesting issue that needs investigation is the type of the index. There are several side effects if a 3-D dense R-tree is used to index a moving point database. Treating time as another dimension in such an index has the following consequences:

– MBRs are continuing to grow larger even if the data do not change their position from one time slot to the other.

- The MBR enlargement is critical to the performance of the index, due to more overlaps among the nodes.
- The index grows rapidly, independently of the mobility of the objects.

We are studying these issues with respect to some important system and database parameters like the size of the database, the mobility of the objects, the number of moving objects, the size of the available buffer space, the complexity of the query execution plan.

## 5 An Example Benchmark

In this section we present some results with respect to the proposed benchmarking environment. The aim is to give some hints about how a benchmark can be generated and how the various iterators are combined together to produce a query execution plan. We illustrate how a two-way join can be modeled and executed in BASIS and give some experimental results regarding the performance evaluation of the various approaches. See [PRS99] for a more detailed treatment of spatial join processing.

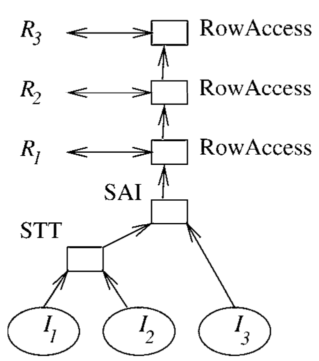
### 5.1 Benchmarking Spatial Join Processing

Fig. 5 illustrates two possible QEPS for processing a two-way join $R_1 \bowtie R_2 \bowtie R_3$, using R*-trees $I_1, I_2$ and $I_3$. For the sake of illustration, we assume that (i) the optimizer tries to use as much as possible existing spatial indices when generating QEPs and (ii) that the 2-way join is first evaluated on the MBRs (filter step) and then on the exact geometry (refinement step, requiring 3 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.

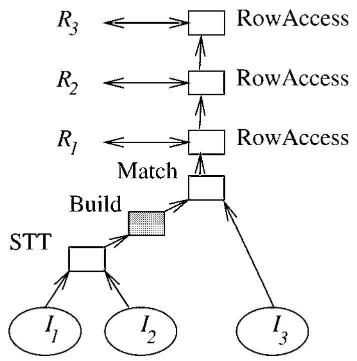
The first strategy (Fig. 5.a) is fully pipelined: an STT (Synchronized Tree Traversal) join is performed as the left-most node, involving $I_1$ and $I_2$. Then an SAI (Scan-and-Index) join is executed for the following join: it takes a pair of `IndexEntry` from the `STT` iterator, and uses one of the MBR as an argument of a window query on $I_3$. The result is a 3-tuple record id: the records are then retrieved with `RowAccess` iterators, one for each relation, to perform the refinement step.

The second strategy (Fig. 5.b) uses instead of SAI a Build-and-Match strategy: since the repeated execution of window queries can be expected to be slow, one can imagine to organize first the result of the STT join (for instance by constructing an index) before performing the second join. Of course the building phase is implemented by a blocking iterator and requires memory space.

These two strategies fit easily in the query execution model of BASIS. Moreover, by considering the whole QEP, one gets an accurate view of the possible shortcomings of each method. For instance, the first strategy is fully pipelined



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



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

**Fig. 5.** Two basic strategies for left-deep QEPs

and enjoys minimal memory requirements while the second one is memory consuming and enforces to wait for the completion of the STT join before proceeding with the second one.

Also, the refinement step could be done prior to the second join if it is expected that the candidate set contains a large number of false hits. By computing the refinement step in lazy mode, as suggested in Fig. 5, the cardinality of intermediate results is larger (because of false hits) but the size of records is smaller. All these "implementation details" are quite useful to determine whether an approach is realistic or not.

The benchmark outlined above allows to compare the relative efficiency of several solutions for the build-and-match strategy. We briefly describe three variants and report the experimental result.

**STJ** (**S**eeded-**T**ree **J**oin)
The first one is the Seeded Tree Join (STJ) [LR98]. This technique builds from an existing R-tree, used as a *seed*, a second R-tree called *seeded R-tree* (i.e, the seeded-tree is built on the result of the STT join (Fig. 5) using $I_3$ as the seeding 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 containing at least one full page are written in temporary files.

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, 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** (**S**ort-**T**ile-**R**ecursive)
The second variant of Build-And-Match algorithm implemented, called Sort-Tile-Recursive (STR) and proposed in [LEL96], constructs on the fly a packed R-tree. 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.

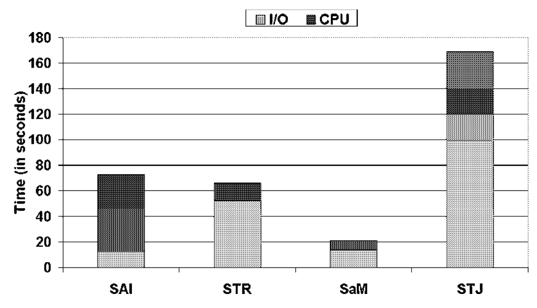
The algorithm is as follows. First, the rectangles from the source are sorted by $x$-coordinate of their centroid. At the end of this step, the size $N$ of the dataset is known: this allows to estimate the number of leaf pages as $P = \lceil N/c \rceil$ where $c$ is the page capacity. The dataset is then partitioned into $\lceil \sqrt{P} \rceil$ vertical slices. The $\lceil \sqrt{P} \rceil .c$ rectangles of each slice are loaded, sorted by $y$-coordinate 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 (see Fig. 2).

**SaM** (**S**ort-**a**nd-**M**atch)
The third Build-And-Match variant called Sort-and-Match (SaM) was proposed in [PRS99]. 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.

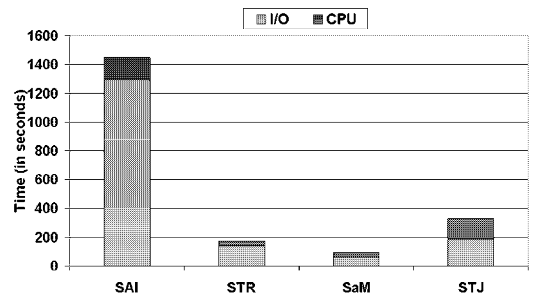
An interesting feature of this algorithm is that, unlike the previous ones, it does not require the entire structure to be built before the matching phase thus saving the flushing of this structure onto disk, resulting in much faster response time.

## 5.2 Two way joins

We now report the results of a performance evaluation with BASIS on the two-way join benchmark previously described. In the sequel, the name of the algorithm denotes the second join algorithm, which takes the result of STT, builds a structure and performs the join with $I_3$.



**40K**



**100K**

**Fig. 6.** Two way joins

We use three datasets from the ENST generator, which simulate respectively *Counties*, *Cities* and *Roads* datasets [GOP$^+$98]. The experiments are performed for the medium dataset size of 40K records and the larger size of 100K. The latter 2 way-join yields 865,473 records, whereas the former produces 314,617 records. A fixed buffer of 500 pages has been chosen.

Fig. 6 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[2] 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.

The results illustrated for STJ are not encouraging. 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. Thus, 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

[2] For a detailed description of the cost calculation and the datasets used see [PRS99].

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.

## 6 Concluding Remarks and Future Work

BASIS is a prototype system aiming at performance evaluation of spatial access methods and spatial queries under a common framework. In the previous sections we discussed the fundamental issues behind the design and implementation of the BASIS system.

Currently we proceed with the extension and improvement of the core BASIS components, as well as with the design and implementation of new ones. We plan to deliver a public release of BASIS to get some feedback from researchers working in spatial and spatiotemporal databases. Our second target is to study the performance of proposed spatiotemporal access methods, using BASIS as the benchmarking platform. Recall that although spatiotemporal access methods have been proposed, there is a lack of a thorough performance evaluation of these methods. Finally, we aim at providing flexibility in query evaluation plans, by incorporating dynamic plans and integrating the modules by means of a friendly user interface.

## Acknowledgments

The authors are very grateful to Prof. Michel Scholl for his helpful comments and suggestions that improved this paper considerably.

## References

[Anon et al.85] Anon et al.: "A Measure of Transaction Processing Power", *Datamation*, Vol.31, No.7, pp.112-118, 1985.

[Are93] W. Aref: "Query Processing and Optimization in Spatial Databases" Technical Report CS-TR-3097, University of Maryland at College Park, 1993.

[BDT83] D. Bitton, D. DeWitt and C. Turbyfill: "Benchmarking Database Systems: a Systematic Approach", *Proceedings 9th VLDB Conference*, pp.8-19, Florence, Italy, 1983.

[BKS90] N. Beckmann, H. P. Kriegel and B. Seeger: "The R$^*$-tree: an Efficient and Robust Method for Points and Rectangles", *Proceedings 1990 ACM SIGMOD Conference*, pp.322-331, Atlantic City, NJ, 1990.

[BKS93] T. Brinkhoff, H-P. Kriegel and B. Seeger: "Efficient Processing of Spatial Joins using R-trees", *Proceedings 1993 ACM SIGMOD Conference*, pp.237-246, Washington DC, 1993.

[BKV98] L. Bouganim, O. Kapitskaia and P. Valduriez: "Memory Adaptive Scheduling for Large Query Execution", *Proceedings 7th CIKM Conference*, 1998.

[Cha95] A. B. Chaudhri: "An Annotated Bibliography of Benchmarks for Object Databases", *ACM SIGMOD Record*, Vol.24, No.1, pp.50-57, 1995.

[DFS98] M. Dumas, M.-C. Fauvet and P.-C. Scholl: "Handling Temporal Grouping and Pattern-Matching Queries in a Temporal Object Model", *Proceedings 7th CIKM Conference*, 1998.

[FBF77] J. H. Friedman, J. L. Bentley and R. A. Finkel: "An Algorithm for Finding the Best Matches in Logarithmic Expected Time", *ACM Transactions on Mathematical Software*, Vol.3, pp.209-226, 1977.

[G94] R. H. Güting: "An Introduction to Spatial Database Systems", *The VLDB Journal*, Vol.3, No.4, pp,357-399, 1994.

[GOP$^+$98] O. Günther, V. Oria, P. Picouet, J-M. Saglio and M. Scholl: "Benchmarking Spatial Joins A la Carte", *Proceedings International Conference on Scientific and Statistical Databases (SSDBM '98)*, 1998.

[Gut84] A. Guttman: "R-trees: a Dynamic Index Structure for Spatial Searching", *Proceedings 1984 ACM SIGMOD Conference*, pp.47-57, Boston, MA, 1984.

[Gra93] G. Graefe: "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys*, Vol.25, No.2, pp.73-170, 1993.

[HNP95] J. M. Helerstein, J. F. Naughton and A. Pfeffer: "Generalized Search Trees for Database Systems", *Proceedings 21st VLDB Conference*, pp.562-573, Zurich, Switzerland, 1995.

[HSW89] A. Henrich, H. W. Six and P. Widmayer: "The LSD-tree: Spatial Access to Multidimensional Point and Non-Point Objects", *Proceedings 15th VLDB Conference*, pp.45-53, Amsterdam, Netherlands, 1989.

[KF93] I. Kamel and C. Faloutsos: "On Packing R-trees", *Proceedings 2nd CIKM Conference*, 1993.

[KF94] I. Kamel and C. Faloutsos: "Hilbert R-tree: an Improved R-tree Using Fractals", *Proceedings 20th VLDB Conference*, pp.500-509, Santiago, Chile, 1994.

[KS95] N. Kline and M. D. Soo: "Time-IT: the TIME Integrated Testbed", September 1995.

[KS97] N. Katayama and S. Satoh: "The SR-tree: an Index Structure for High-Dimensional Nearest Neighbor Queries", *Proceedings 1997 ACM SIGMOD Conference*, pp.369-380, May 1997.

[LEL96] S. Leutenegger, J. Edgington and M. Lopez: "STR: a Simple and Efficient Algorithm for R-tree Packing", *Proceedings 12th IEEE ICDE Conference*, 1996.

[LRB$^+$97] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki and K. Wenger: "DEVISE: Integrated Querying and Visual Exploration of Large Datasets", *Proceedings 1997 ACM SIGMOD Conference*, 1997.

[LT92] R. Laurini and D. Thomson: "*Fundamentals of Spatial Information Systems*", Academic Press, London, 1992.

[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*, Vol.10, No.1, 1998.

[ND98] B. Nag and D. DeWitt: "Memory Allocation Strategies for Complex Decision Support Queries", *Proceedings 7th CIKM Conference*, 1998.

[PRS99] A.N. Papadopoulos, P. Rigaux and M. Scholl: "A Performance Evaluation of Spatial Join Processing Strategies", *Proceedings International Conference on Large Spatial Databases (SSD'99)*, Honk-Kong, China, 1999.

[RKV95] N. Roussopoulos, S. Kelley and F. Vincent: "Nearest Neighbor Queries", *Proceedings 1995 ACM SIGMOD Conference*, pp.71-79, San Jose, CA, 1995.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith: "The Sequoia 2000 Storage Benchmark", *Proceedings 1993 ACM SIGMOD Conference*, pp.2-11, Washington DC, 1993.

[SRF87] T. Sellis, N. Roussopoulos and C. Faloutsos: "The R$^+$-tree: a Dynamic Index for Multidimensional Objects", *Proceedings 13th VLDB Conference*, pp.507-518, Brighton, UK, 1987.

[Tiger94] Bureau of the Census: "Tiger/line files", Washington DC, 1994.

[TPCD95] Transaction Processing Performance Council TPC. *TPC Benchmark D Specification*, version 1.1 edition, December 1995.

[TPCC97] Transaction Processing Performance Council TPC. *TPC Benchmark C Specification*, revision 3.3.2 edition, June 1997.

[Val87] P. Valduriez: "Join Indices", *ACM Transactions on Database Systems*, Vol.12, No.2, pp.218-246, 1987.

[XHL90] X. Xu, J. Han and W. Lu: "RT-Tree: an Improved R-tree Index Structure for Spatiotemporal Databases", Proceedings 4th Symposium on Spatial Data Handling (SDH'90), pp.1040-1049, 1990.

## Appendix A - Fundamental Classes

BASIS is composed of a set of classes. Some of the most important ones are illustrated in Table 2, with a short description.

## Appendix B - Iterators

We give in this appendix a description of the iterators. Each iterator comes with three functions: `open`, `next` and `close`: `open` prepares each input source and allocates the necessary system resources, `next` runs one iteration step of the algorithm and produces a new record and `close` ends up the iteration and relaxes resources.

To illustrate this, consider the trivial example of the `FileScan` iterator which sequentially accesses a data file. It is implemented as follows: (i) `Open` opens the data file, and sets a cursor to the beginning of the file, (ii) `Next` reads the file page addressed by the cursor, returns the current record to the father node iterator, and sets the cursor to the next record, (iii) `Close` closes the file. The following table summarizes the iterators used in this study.

| **Class** | **Description** |
|---|---|
| CStorageManager | Abstraction of a simple page-oriented storage manager |
| CBufferManager | Implementation of a page-oriented buffer manager. Manages a number of buffer pools. |
| CFileManager | Manages a number of BASIS files, that comprise the database. |
| CFile | Abstraction of a file. |
| CStorable | Abstract base class for objects that need to be *serialized* on disk. |
| CRecord | Abstraction of a fixed-length record. |
| CRecordType | Allows the creation of records with different attribute types. |
| CPage | Page abstraction. A CPage object can contain anything that inherits from CStorable. |
| CSAM | Abstract base class for spatial access methods. |
| CIndex | Abstract base class for indexes. |
| CDatafile | Manages a database file which contains a dataset. |
| CRtree | Inherits from CIndex and implements an R-tree. |
| CRstar | Inherits from CIndex and implements an R$^*$-tree. |
| CGrid | Inherits from CIndex and implements a Grid-file access method. |
| CApprox | The record part which is used for indexing. |
| CRect | Abstraction of an N-D rectangle. |
| CPoint | Abstraction of an N-D point. |
| CLineSegment | Abstraction of an N-D line segment. |
| CTrajectory | Abstraction of an N-D trajectory. Contains a collection of line segments. |
| CStatistics | Contains information about the performance of a method (e.g. disk accesses). |
| CIterator | Used to construct query execution plans. |

**Table 2.** Fundamental classes of BASIS.

| | `open` | `next` | `close` |
|---|---|---|---|
| `FileScan` | Open the file; | Retrieve the record; Advance the cursor; | Close the file |
| `RowAccess` | `Open` input; Open the file; | Call `next` on input; Read the record; | `Close` input Close file |
| `Sort` | `Open` input *Repeat*: fill the buffer from input; sort the buffer; flush in runs. Merge until only one step is left. `Close` input. | Process one merge step | Release the buffer and tmp files |
| `SegSort` | `Open` input. *Prepare*: put records in the buffer from input; sort on $R$ record ids; read records from $R$; sort on $S$ record ids. | Take the next pair in the buffer; read the record from $S$ Buffer empty? Then execute *Prepare* | `Close` input |
| `STT` | Open R-trees; Init. the paths in each R-tree; Pin and sort pages as required. | Get next pair of entries; When a page is scanned: unpin, get the next one, sort and pin. | `Close` both inputs |
| `STJ` | `Open` input; Build the seeded tree from input. `Close` left input Init. as in STT. | Same as STT | `Close` right input |
| `STR` | `Open` input; Build the STR from input; `close` left input; Init. as in STT | Same as STT | `Close` right input |
| `SaM` | `Open` input; Sort on $x$. Sort the first slice on $y$. Get the first matching leaf in the R-tree; sort it (on $y$) | Get next pair of entries. *Event.*: get the next leaf in the R-tree; sort it *Event.*: get the next slice on $y$ | `Close` both inputs |

**Table 3.** Description of iterators.