Multiple Entry Indexing and Double Indexing [*]

Victor Teixeira de Almeida, Ralf Hartmut Güting, and Christian Düntgen
Database Systems for New Applications, Faculty of Mathematics and Computer Science
University of Hagen
D-58084 Hagen, Germany
`{victor.almeida, rhg, christian.duentgen}@fernuni-hagen.de`

Abstract

Traditional indexing techniques are not well suited for complex data types such as spatial, spatio-temporal, and multimedia data types, where an instance is a composite of multiple components. In this paper we propose two indexing techniques that allow the parts of a composite object to be indexed separately, called multiple entry indexing and double indexing. We present the implementation of these approaches in the SECONDO extensible database system. The improvements in terms of performance of both approaches presented in this paper shown in an experimental evaluation.

1 Introduction

Non-conventional database systems support complex data types such as spatial, spatio-temporal, and multimedia data types such as lines, regions, moving objects, texts, images, audio data, etc. One common property of such data types is that they have variable length size, but their size is unpredictable, it can be very small, e.g. a 16 colors 16x16 pixels icon, or very big, e.g. a high resolution satellite image. Another important property of these data types is that sometimes they contain multiple components. A line is composed by a set of segments, a moving object by a set of units with linear movement, an audio by a set of tracks, etc.

Indexing mechanisms are crucial for efficient query processing coping with these complex data types. Approximations are used to index such attributes, e.g. minimum bounding rectangles (MBR) for spatial data types. We will use spatial and spatio-temporal data types as examples in the rest of this section. The MBR of an object is the minimum rectangle aligned with the coordinate axes that totally encloses the object.

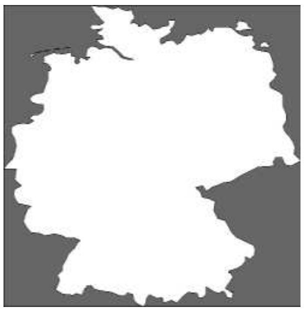


Figure 1: The Minimum Bounding Rectangle (MBR) of Germany.

Figure 1 shows the region of Germany and its MBR. The dark color represents the so-called dead space, which is the difference between the area of the MBR and the object area. It is well

[*]This work was partially supported by the grant Gu 293/8-2 from the Deutsche Forschungsgemeinschaft (DFG), project "Datenbanken für bewegte Objekte" (Databases for Moving Objects)

known that the amount of dead space in a data set deteriorates the performance of a query on such indexes.

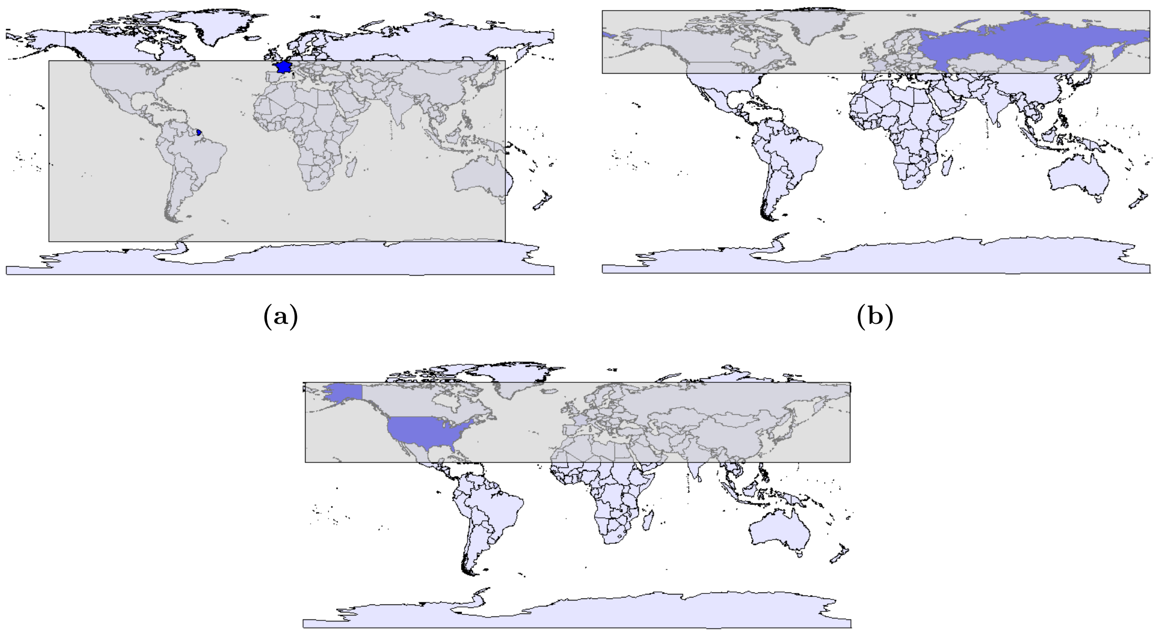


Figure 2: The MBRs of (a) France, (b) Russia, and (c) USA.

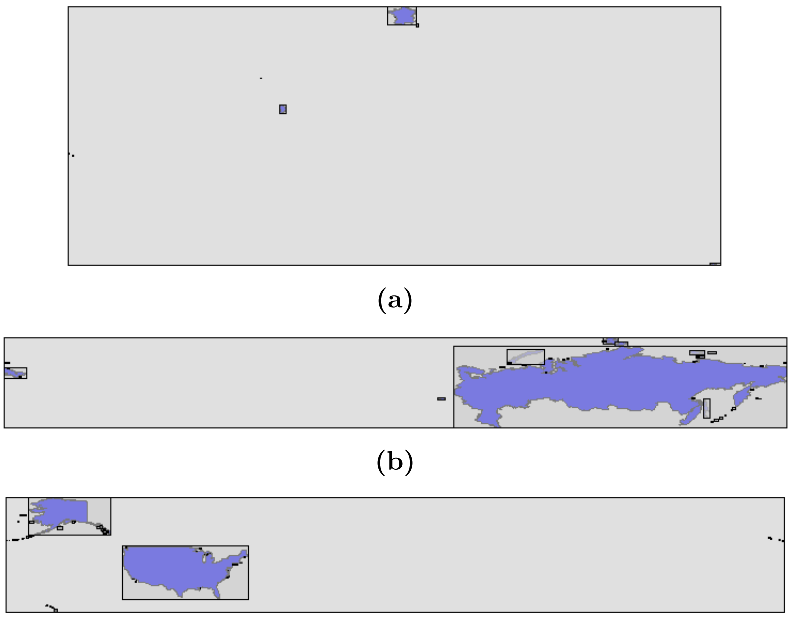
In some cases, the amount of dead space can be unacceptable. Figure 2 shows some of these cases using a data set containing the world countries as regions, which we call the world data set. The regions of France, Russia, and the USA contain several components and a huge dead space is introduced by their MBRs, shown in Figure 2(a), 2(b), and 2(c), respectively.

The data set contains 239 tuples and the MBR of France has intersection with 207 other MBRs, but the region of France just intersects 15 other regions. The following query: "Return the neighbor countries of France" can use a spatial index to improve its efficiency, i.e. before checking the intersection between the real representation of the countries, a search in the spatial index is done to retrieve the entries that have MBR intersection with France's MBR. However, in this case, this step is not very restrictive, given that it returns 86,61% of the data set.

The dead space would be significantly reduced if every component is considered separately by the index, which means every component of a country is separately indexed. We call this approach *multiple entry indexing*, because every tuple in the data set may contain several entries in the index. Figure 3 shows the MBRs of each component of (a) France, (b) the USA, and (c) Russia.The margins of each figure represent the MBR of the whole country. One can see in these examples on the one hand that the dead space reduction is very significant in these cases, whereas, on the other hand the number of entries in the index increases.

Table 1 shows the improvement achieved to answer the query: "Find the countries that intersect country X" where we used France, USA, and Russia as X. The first column presents the number of MBR intersections using the whole country MBRs, the second and third ones present the number of MBR intersections using component MBRs without and with an additional step to remove duplicates, respectively. The fourth column shows the expected result of the query, i.e. the number of intersections using the real representation of the objects. We can see that the number of computations of intersections using the real representation of the countries is reduced by approximately a factor of 12 for France, 27.5 for the USA, and 2.5 for Russia using the multiple entry indexing approach with duplicate removal.

With multiple entry indexing it is possible, for these composite data types, to perform queries


(a)

(b)

(c)

Figure 3: The MBRs of each component of (a) France, (b) Russia, and (c) USA.

Table 1: Intersections between all countries and France, USA, and Russia.

Country	MBR intersections	Component MBR intersections (with duplicates)	Component MBR intersections (without duplicates)	Real intersections
France	207	21	17	15
Russia	110	42	4	2
USA	60	33	25	13

more efficiently. Obviously, not in every case multiple entry indexing is better than traditional approaches, because it needs an additional duplicate removal step after the execution of the query in the index. The more selective the query the bigger the gain we achieve with multiple entry indexing.

Multiple entry indexing is similar to partial indexes ([9, 10]) in the PostgreSQL DBMS. However, with partial indexes it is possible to restrict the tuples in the relation to be indexed to a subset with the help of a SQL query but it is not allowed that the index contains multiple entries for each tuple of the corresponding relation.

However, there is still place for optimization. In traditional query processing with indexes, every leaf entry in the index points to the corresponding tuple of that entry and, once the entry's approximation satisfies the query predicate, the object inside the tuple is retrieved and then the query predicate is applied to it. In our case, where objects can contain several components, we also want to avoid reading the whole object, but only the components of the object that satisfied the query predicate. With this approach the gains are twofold: we do not read the whole object (I/O), which can span into several disk pages, and we do perform the predicate in complex objects only containing the components that satisfied the approximation test (CPU).

The technique is then the following: when inserting an entry for a component into the index we carry additional information about the position of that component in the composite object. We decided to keep this information as simple as possible because it consumes valuable space in the index leaf entries. To achieve this goal, an interval is stored together with the leaf entries in the index. This interval stores the starting and ending position of the component in the whole object. In query processing, after performing the query in the index, the result is then sorted by tuple identifiers and by the intervals. In order to retrieve the tuples, the set of intervals of the same tuple is used to retrieve only a restricted portion of the complex object. We call this approach *double indexing*.

In our example of the world data set, if we check whether the USA and Russia intersect each other using the double indexing approach dividing the regions by component, only a few islands of the USA MBRs intersect the MBR of the bigger component of Russia (Figure 4). In this case, only a few pages will be read for the USA components (maybe only one) and the complexity of the intersection predicate is lowered given that the representation of the reduced USA instance is much smaller.

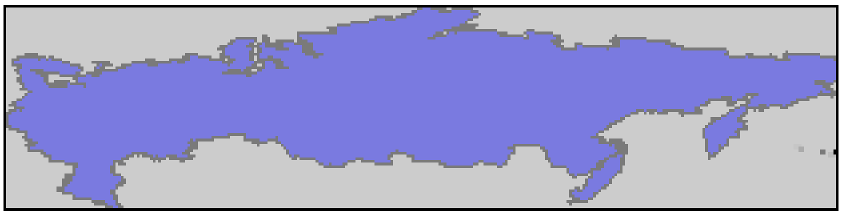


Figure 4: MBR intersection between the components of the USA and Russia.

In this introduction, we used a spatial data set to demonstrate the motivation to our work. However, this approach is general and is not limited to spatial databases. Every composite data type can be indexed using the two approaches presented in this paper. As an example, the works in [7, 8] present an approach to splitting a moving object trajectory into pieces before indexing it, without considering that this is not possible using traditional indexing approaches.

In this paper we present an integrated approach in the Secondo extensible database system implementing both multiple entry indexing and double indexing. The paper is organized as follows: Section 2 presents a brief overview of Secondo. Section 3 presents the operators for both indexing approaches in the paper. In Section 4 we present an experimental evaluation comparing the techniques presented in this paper against traditional indexing approaches. Finally, Section 5 concludes the paper presenting an overview of future work.

2 Secondo Overview

In order to explain the implementation of our indexing approach proposed in this paper, we first need to present a brief overview of the Secondo extensible database system, which is done in this section. The goal of Secondo is to provide a "generic" database system frame that can be filled with implementations of various DBMS data models. For example, it should be possible to implement relational, object-oriented, temporal, or XML models and to accommodate data types for spatial data, moving objects, chemical formulas, etc.

Secondo has been demonstrated at some conferences, such as ICDE'05 ([1,6]) and MDM'06 ([2]).

The Secondo system consists of three major components shown in Figure 5:

- The Secondo kernel implements specific data models, is extensible by algebra modules, and provides query processing over the implemented algebras. It is written in C++.
- The optimizer provides as its core capability conjunctive query optimization, currently for a relational environment, and also implements the essential part of SQL-like languages. It is written in Prolog.

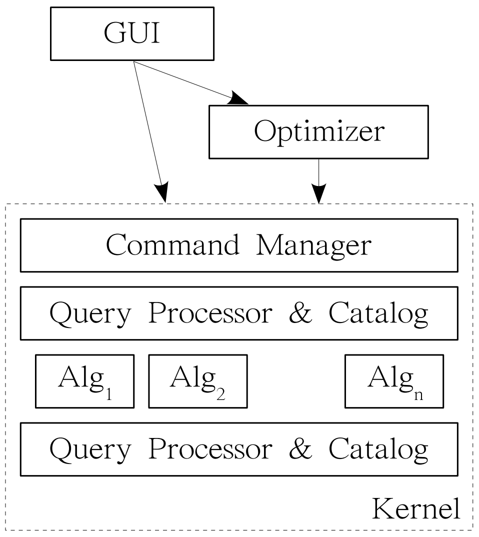


Figure 5: The SECONDO architecture.

- The graphical user interface (GUI) is an extensible interface for an extensible DBMS such as SECONDO, where new data types or models can provide their own viewers or extend an existing viewer by display methods. It is written in Java.

2.1 The Kernel

A very rough description of the architecture of the SECONDO kernel is shown in the bottom of Figure 5. A data model is implemented as a set of data types and operations. These are grouped into *algebras*. For example, there is an algebra called *Relational Algebra* with tuples and relations as data types and operations like projection or hashjoin. Traditional indexing techniques are supported on SECONDO with algebras for B-trees and R-Trees.

The kernel can evaluate a query plan, also called an executable query, or a query at the executable level, which is just a term of the implemented algebras. Query processing is performed as follows: the Command Manager receives an executable query, parses it and passes the result to the Query Processor. The Query Processor then evaluates the query by building an operator tree and then traversing it, calling operator implementations from the algebras. More details about this process can be found in [3]. SECONDO objects are stored (and retrieved) by the Storage Manager into a database and managed by the Catalog.

A collection of algebras is available in the SECONDO Kernel, such as the *Standard Algebra* containing standard data types like *bool*, *int*, *real*, and *string*; the *Relational Algebra* containing the data types for tuples (*tuple*) and relations (*rel*); the *BTree Algebra* and the *RTree Algebra* containing implementation of B-Trees and R-Trees indexes, respectively; the *DateTime Algebra* with the implementation of the *instant* and *duration* data types; the *Spatial Algebra* containing 2-dimensional data types like *point*, *line*, and *region*; the *Temporal Algebra* providing moving object data types (e.g. *moving*(*point*)).

An important feature of SECONDO is that it offers a specific concept for the implementation of persistent attribute data types, where they are represented as a *root record* and can contain some *database arrays*. Database arrays are basically persistent arrays of elements with fixed size, and are implemented on top of a concept called FLOB (Faked Large Object) described in [4], which means that they are automatically either represented inline with the tuple representation, or outside in a separate list of pages, depending on their sizes. FLOBs are read paged from disk. As an example, the *line* data type could contain some aggregate information in the root record such as its length, its MBR, the number of disconnected components, etc. and a database array of line segments.

2.2 The Optimizer

The optimizer provides as its core functionality cost-based optimization of conjunctive queries. That is, it receives a set of relations together with a collection of selection and join predicates, and produces a plan. It employs a novel algorithm for query optimization described in detail in [5], based on shortest path search through a predicate order graph. It is written in Prolog, using the SWI-Prolog system.

On top of the conjunctive query optimizer, the essential parts of an SQL-like language have been implemented. The SQL notation was slightly adapted so that queries can be written directly as PROLOG terms. Since SECONDO is extensible by algebras, some of them containing complex data types, we have the following requirements:

- Selectivity estimation must work for complex data types and an extremely large set of operations. The traditional histogram-based approach does not scale to this case.
- Operations can be expensive; hence expensive predicates must be supported in optimization.

SECONDO provides selectivity estimation by sampling; for each relation a small materialized sample is kept. Unknown selectivities are determined in advance by sending selectivity queries to the kernel before starting the proper optimization process; they are then stored for later use.

The cost for expensive predicates is determined as well in the execution of the selectivity queries on samples by measuring the actual execution time, subtracting overhead.

The beauty of this scheme is that optimization works to a large extent automatically without manual work when a new algebra with non-standard types is added. What has to be provided manually are optimization rules for adding specialized indexes, and possibly syntax rules for operations (the latter is very easy).

2.3 The User Interface

A visualization of query results is possible in the graphical user interface called *Javagui* of the SECONDO system. Javagui communicates with the system kernel and the optimizer via TCP/IP. It can be extended by viewers. Each viewer can display a set of different data types. In this way, Javagui is able to display each data type implemented in the system kernel.

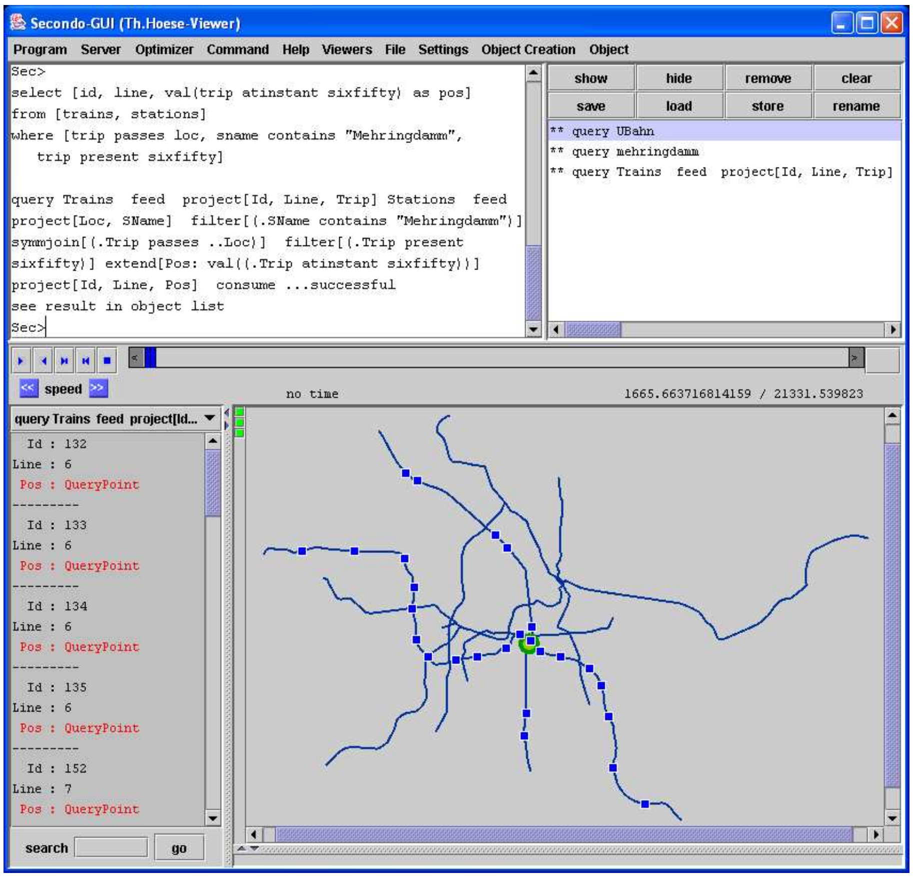
The user interface consists of three major parts (see Figure 6), namely the command area (top-left), the object manager (top-right), and an area containing the current viewer (bottom).

In the command area, the user can input queries and commands controlling Javagui. Javagui recognizes whether a query is given at the executable level or in the syntax of the optimizer. If the query is in optimizer syntax, Javagui sends it to the optimizer and receives a plan at the executable level. This plan is sent to the system kernel. The result of a query is delivered in a generic format based on nested list structures to the object-manager. It stores the result of the query, selects a viewer able to display the result, and finally it transfers the query result to this viewer for further processing.

The HoeseViewer (named by its author) is a fairly sophisticated viewer for spatial and spatio-temporal data. This viewer can be extended for displaying further data types using display classes.

3 Indexing

In this section, we give detailed information about the implementation of the indexing approach presented in this paper. As said in the previous sections, SECONDO already has algebras for


Secondo-GUI (Th.Hoese-Viewer)
Program Server Optimizer Command Help Viewers File Settings Object Creation Object
show hide remove clear
save load store rename
speed
search go
query Trains feed project[Id...

```
Sec>
select [id, line, val(trip atinstant sixfifty) as pos]
from [trains, stations]
where [trip passes loc, sname contains "Mehringdamm",
   trip present sixfifty]

query Trains  feed  project[Id, Line, Trip] Stations  feed
project[Loc, SName]  filter[(.SName contains "Mehringdamm")]
symmjoin[(.Trip passes ..Loc)]  filter[(.Trip present
sixfifty)] extend[Pos: val((.Trip atinstant sixfifty))]
project[Id, Line, Pos]  consume ...successful
see result in object list
Sec>
```

```
** query UBahn
** query mehringdamm
** query Trains  feed  project[Id, Line, Trip]
```

```
no time          1665.663716814159 / 21331.539823
```

```
 Id : 132
Line : 6
 Pos : QueryPoint
---------
 Id : 133
Line : 6
 Pos : QueryPoint
---------
 Id : 134
Line : 6
 Pos : QueryPoint
---------
 Id : 135
Line : 6
 Pos : QueryPoint
---------
 Id : 152
Line : 7
 Pos : QueryPoint
```

Figure 6: The SECONDO graphical user interface.

indices such as B-trees and R-trees. These algebras contain data types for such indices and some operators mainly for creating the index and querying it. We focus this paper on the R-tree indexing, since we use spatio-temporal data sets in our experiments. Everything that is presented here can be, and actually is, extended to indexing of standard data types with B-trees.

During the operators syntax description, we will follow our discussion with some examples in order to clarify the explanations. In the examples, we use the Berlin data set, graphically seen in Figure 6, which contains several relations with spatial objects in the city of Berlin, Germany — such as streets, underground train lines, green and water areas, sightseeing spots, restaurants, etc. — and a relation containing several lines of underground trains as moving points. This relation containing moving points is the one that will be mainly used in our examples and is described as

```
Trains( Id: int, Line: int, Up: bool, Trip: mpoint )
```

where *Id* is a numeric identifier (primary key), *Line* contains the line number of the train, *Up* identifies the direction of the movement of the train in the line, and *Trip* contains the moving point object.

The presentation of the operators is divided into three steps containing the operators to create and query standard indexes, the operators for multiple entry indexing, and finally the operators for double indexing.

3.1 Operators for Traditional Indexing

A traditional query processing with the help of an index runs in the following steps:

1. Send the query condition to an index and receive some result candidates
2. Retrieve the tuples pointed to by the candidates from their respective relations
3. Apply the query condition to these tuples and return the result (this step is needed if the index search cannot decide the predicate, as in filter and refine techniques).

Indices are directly connected to relations and every tuple of the relation has a corresponding entry in the index. Otherwise, the query processing will not work.

The operator called **createrrtree** which performs the index creation needs a relation as one of the arguments and the attribute on that relation that needs to be indexed as the second argument. In the case of our examples, the attribute must be of a spatial type, e.g. $\underline{point}$, $\underline{points}$, $\underline{line}$, or $\underline{region}$, or of a spatio-temporal type, e.g. $\underline{mpoint}$ ($\underline{mpoint}$ is the abbreviation of $\underline{moving}(\underline{point})$ which is the data type for moving points). MBR approximations of the values of this attribute are indexed in the R-Tree. Here is the description of the **creatertree** operator:

$\underline{rel}(\underline{tuple}(X)) \times attrname \quad \rightarrow \underline{rtree}(\underline{tuple}(X)\ attrtype\ Std)$ **creatertree**

One should note that this operator creates a standard R-tree, which can be seen in the third parameter in the R-tree data type description (*Std*).

Using the *Trains* relation we can then create a standard R-tree index using the following command:

```
let trains_Trip =
  Trains creatertree[Trip];
```

This operation creates a 3D R-Tree indexing the 3D MBRs (2D MBR + time) of the attribute *Trip* of the *Trains* relation.

The following operation, called **windowintersects**, is available for querying:

$\underline{rtree}(\underline{tuple}(X)\ attrtype\ Std) \times \underline{rel}(\underline{tuple}(X)) \times \underline{rect}\text{<dim>}$
$\quad \rightarrow stream(\underline{tuple}(X))$ **windowintersects**

This operator receives an R-Tree, a relation and a <*dim*>-dimensional query rectangle (window) as arguments — where *dim* is the dimensionality of the indexed attribute type —, searches in the R-Tree all entries that intersect the query rectangle and retrieves the corresponding tuples in the relation, returning them as a stream. Streams of tuples are used internally in the SECONDO system in order to pipeline the operators of the relational algebra.

The **windowintersects** operator only performs the steps (1) and (2) of the query processing scheme for retrieving the tuples given the approximation in the index. Step (3) of performing the query predicate on the candidate tuples must be executed later. The following query example counts the MBRs that intersect the MBR of the Mehringdamm Station (object *mehringdamm*):

```
query trains_Trip Trains
  windowintersects[ bbox(mehringdamm) ]
  count;
```

If we want to retrieve the objects that really pass the Mehringdamm Station, the predicate filter needs to be added after the index query.

```
query trains_Trip Trains
  windowintersects[ bbox(mehringdamm) ]
  filter[ .Trip passes mehringdamm ]
  count;
```

This query completely represents all steps (1), (2), and (3) of traditional query processing scheme stated in the beginning of this sub-section.

3.2 Operators for Multiple Entry Indexing

To support multiple entries in the index we must get rid of the relation argument in the **creatertree** operator and allow a stream of tuples as input. This stream of tuples must contain one and only one attribute of type *tid*, which stores tuple identifiers.

stream(*tuple*([X, id: *tid*])) × *attrname*
  → *rtree*(*tuple*(X) *attrtype Std*)   **creatertree**

The type of this R-Tree created is the same as the one created in Section 3.1 (*Std*). The following command creates the same R-Tree as in the previous example.

```
let trains_Trip =
  Trains feed addid creatertree[Trip];
```

Here we make use of the **feed** and **addid** operators. **feed** converts a relation into a stream of tuples and **addid** extends the tuple with its tuple identifier, an internal number that uniquely identifies a tuple in the relation.

With this new way of creating an R-tree index we have much more flexibility. The following example creates an index only for those objects that pass the Mehringdamm Station, which is not possible to be done with the traditional R-Tree creation operator.

```
let trains_Trip =
  Trains feed addid
  filter[ .Trip passes mehringdamm ]
  creatertree[Trip];
```

The following operator, called **windowintersectsS** ('S' comes from *stream*), allows us to query a multiple entry index without directly specifying the relation, from which the tuples should be retrieved. The result is a stream of tuples containing only one attribute of *tid* type containing the tuple identifiers of the entries in the index that satisfy the query condition.

rtree(*tuple*(X) *attrtype Std*) × *rect*<*dim*>
  → *stream*(*tuple*([*id*: *tid*]))   **windowintersectsS**

In order to later retrieve the tuples from the corresponding relation, the **gettuples** operator is provided. This operator receives a stream of tuples containing an attribute of type *tid* with tuple identifiers and a relation, and retrieves the tuples from the argument relation appending their attributes (Y) to the end of the tuple received as argument (X). The attribute with the tuple identifiers is removed from the resulting tuple type. The result of this operator is still a stream of tuples.

stream(*tuple*([X, *id*: *tid*])) × *rel*(*tuple*(Y))
  → *stream*(*tuple*([X, Y]))   **gettuples**

As an example, with this new query operator, to simply count the number of MBR intersections we have between the trips and the Mehringdamm Station, we do not need to load the tuples from the *Trains* relation as follows:

```
query trains_Trip
  windowintersectsS[ bbox(mehringdamm) ]
  count;
```

If we want to know the trips that really passed the Mehringdamm Station, the query is then

```
query trains_Trip windowintersectsS[ bbox(mehringdamm) ]
  Trains gettuples
  filter[ .Trip passes mehringdamm ]
  count;
```

However, this query does the same as the standard one. The power of this approach comes from the fact that we have more flexibility on indexing and retrieving tuples.

As an example of this flexibility we will now create an R-tree containing entries for every piece of movement of the trips in the *Trains* relation. One should note that now we (possibly) have multiple entries for the same tuple of the *Trains* relation in the index.

```
let trains_Trip_unit =
   Trains feed addid
   extendstream[Unit:  units(.Trip)]
   creatertree[Unit];
```

For this query, we make use of the operators **extendstream** and **units**. The **units** operator takes a moving object, a *mpoint* in this example, and returns a stream of temporal units, which are the linear pieces of the movement, in this case of type *upoint*. The operator **extendstream** combines the tuples coming from the outer stream with the ones coming from the inner stream. With the combination of these two operators we expand the units inside the temporal data type, indexing them into the resulting R-tree.

Now, the query to find the trips that passed the Mehringdamm Station is

```
query trains_Trip_unit
   windowintersectsS[ bbox(mehringdamm) ]
   sort rdup
   Trains gettuples
   filter[ .Trip passes Mehringdamm ]
   count;
```

Note that we added a duplicate removal after the index query (step (2) in the new query processing scheme below), with the usage of the **sort** and **rdup** operators, to eliminate possible duplicate results for the same tuple. The query processing scheme can now be stated as

1. Send the query condition to the index and receive some result candidates
2. Remove duplicates
3. Retrieve the tuples pointed to by the candidates from their respective relations
4. Apply the query condition to these tuples and return the result (this step is needed if the index search cannot decide the predicate, as in filter and refine techniques).

3.3 Operators for Double Indexing

In the last sub-section, we showed how to change the traditional query processing with the help of an index to allow multiple entries in the index. However, there is still place for optimization, because we still need to retrieve the entire tuples in order to perform the last query condition check.

It would be perfect if we could load only the parts of the tuples that conform with the query condition in the index retrieval. This is done with our last approach called double indexing. With double indexing, we are not only able to store multiple entries in the index but also to store an interval for every entry in order to, when retrieving the tuples, restrict the object loading only to the components that are of interest for the query, i.e. to retrieve only the components that satisfy the query predicate in the index retrieval.

In order to achieve this, the **creatertree** operator is again modified to receive the *low* and *high* attributes in the stream of tuples, which represent the interval.

$$\mathit{stream}(\underline{\mathit{tuple}}([X,\ \mathit{id}\text{: }\underline{\mathit{tid}},\ \mathit{low}\text{: }\underline{\mathit{int}},\ \mathit{high}\text{: }\underline{\mathit{int}}]\ ))\times \mathit{attrname}$$
$$\rightarrow \underline{\mathit{rtree}}(\ \underline{\mathit{tuple}}(X)\ \mathit{attrtype}\ \underline{\mathit{Dbl}}) \qquad \textbf{creatertree}$$

One should note that the R-Tree index description is now different, i.e. of type *Dbl* (from double).

Following our examples using the Berlin data set, we can create an R-tree index with one entry for every piece of movement of the trips in the *Trains* relation.

```
let trains_Trip_unit =
  Trains feed addid
  filter[seqinit(0)]
  extendstream[Unit:  units(.Trip)]
  extend[Box:  bbox(.Unit), Low:  seqnext()]
  extend[High:  .Low]
  project[Box, TID, Low, High]
  creatertree[Box];
```

In this first index, the values of *high* and *low* are always the same, informing to the index that there is only one entry indexed by this value in the trip. The operator **seqinit** initializes a counter to its argument's value and return 'TRUE', while the operator **seqnext** returns the counter's current value and post-increment it.

The next example indexes every group of 5 units of the trips in the *Trains* relation. We are not going to explain every detail on this example since it uses complex operators (with parameter functions) of the Relational Algebra. We ask the reader to believe that this query below does index every group of 5 units and setting the intervals to [0, 4], [5, 9], ...

```
let trains_Trip_group5 =
  Trains feed addid
  filter[seqinit(0)]
  projectextendstream[TID; Unit:  units(.Trip)]
  extend[Grp:  seqnext() div 5, Box:  bbox(.Unit)]
  groupby[TID, Grp; LargeBox:  group feed aggregateB[Box; fun( r1:rect3, r2:rect3 ) r1
union r2; [const rect3 value undef]], N: group count]
  extend[Low:  .Grp * 5]
  extend[High:  .Low + .N - 1]
  project[LargeBox, TID, Low, High]
  creatertree[LargeBox];
```

The same **windowintersectsS** operator as for multiple indexing is available, but it returns not only the tuple identifier, but also the *low* and *high* values for every entry in the candidate result.

$\underline{rtree}(\underline{tuple}(\mathrm{X})\ attrtype\ Dbl) \times \underline{rect}{<}dim{>}$
$\quad\rightarrow stream(\mathrm{tuple}([id\text{: }\underline{tid},\ low\text{: }\underline{int},\ high\text{: }\underline{int}]\ ))$ **windowintersectsS**

To later retrieve the tuples, a new version of the **gettuples** operator is provided, called **gettuplesdbl**. This operator expects a tuple in the format that is returned from the latter operator, but it also expects that the tuples are sorted by *tid*, *low*, and *high* in ascending order.

The operator combines all intervals of the same tuple identifier into a set of intervals. Then it retrieves the tuple corresponding to the tuple identifier, but restricts the reading of the attribute named by the operator's third argument to only those components given by this set of intervals mentioned above.

$stream(\underline{tuple}([\mathrm{X},\ id\text{: }\underline{tid},\ low\text{: }\underline{int},\ high\text{: }\underline{int}\ ]))$
$\quad\times\ \underline{rel}(\underline{tuple}(\mathrm{Y}))\times attrname$
$\quad\rightarrow stream(\underline{tuple}([\mathrm{X},\ \mathrm{Y}]))$ **gettuplesdbl**

The query that counts the number of objects that pass through the Mehringdamm Station can then be now re-written as

```
query trains_Trip_unit (or trains_Trip_group5)
  windowintersectsS[ bbox(mehringdamm) ] sort
  Trains gettuplesdbl[Trip]
  filter[ .Trip passes Mehringdamm ]
```

```
count;
```

The tuples retrieved by the **gettuplesdbl** are restricted to the components that satisfy the query predicate in the index. This approach improves the performance of the query processing in two ways. First, it allows one to retrieve only parts of the object, which means it reduces the number of disk accesses. Second, as far as the object passed to the last filtering condition is (possibly) reduced, the filtering condition step can also be performed faster, which means less CPU time.

The query processing scheme of the last two sub-sections can now be stated as

1. Send the query condition to the index and receive some result candidates
2. Sort the entries on the candidate set by *id*, *low*, and *high* attributes
3. Retrieve the tuples pointed to by the candidates from their respective relations, restricting the query attribute to only those parts in the intervals represented by the *low* and *high* values.
4. Apply the query condition to these tuples and return the result (this step is needed if the index search cannot decide the predicate, as in filter and refine techniques).

4 Experimental Evaluation

In this section we aim to show the advantages of both multiple entry and double indexing approaches. We use two data sets for this purpose:

- **TrainsL.** We translated the Berlin database explained in Section 3 five times in all directions: x, y, and *time*. We then have a database that is 125 times larger than the Berlin database. The translated *Trains* relation we call *TrainsL*, where 'L' comes from *large*. This relation contains a relatively large number of tuples with moving objects (*TripL* attribute) of small trajectories, consequently of small size.
- **U1lin.** This data set is similar to *Trains* relation in the Berlin data set explained in Section 3. The *U1lin* relation contains only one train line (*U1*) with 13 trains performing 11 two-way trips per day for 6 months. The trains with even identifiers stop on weekends and the ones with odd identifiers run on weekends as they do on weekdays. This relation contains a small number of tuples with moving objects (*TripL* attribute) with large trajectories.

Some statistics about both data sets are provided in the Table 2.

Table 2: Statistics about the data sets.

	TrainsL	U1lin
Size in MBytes	673.80	429.56
# of tuples	70,250	13
Avg. # of units per tuple	82.025	309,355
Avg # of pages per tuple	3	8594

For creating the indexes we have the following variables:

- **dimensionality**: spatial (dS), temporal (dT), or spatio-temporal (dST)
- **entry type**: one entry per object (eObj), one entry per temporal unit (eUnt)

- **index type**: standard (iStd) and double index (iDbl)

Let us use one of the combinations to show how we name the indexes in the rest of this section. The spatio-temporal double index with one entry per unit for the relation *TrainsL*, attribute *TripL*, is called *trainsl_TripL_dST_eUnt_iDbl*. One should note that not all combinations make sense, e.g. double indexing with one entry per object. We call simply *trainsl* the approach that sequentially scans the whole relation directly applying the query predicates, bypassing the index filtering step.

4.1 Temporal queries

For temporal queries we used the **atinstant** operator receiving a moving point (*mpoint*) and an instant i as arguments. This operator restricts the movement to the position at time instant i. We performed the following query with five different values of i: the first instant of the data set, the last instant, the first and third quartiles, and the middle instant.

The query in SQL-like format is written below. We use a SQL-like format to write the query to avoid presenting all queries for all approaches in the SECONDO executable format.

```
select val(trip atinstant i) as pos
  from trainsl
  where trip present i;
```

We added the predicate with the operator **present** in the *where* clause in order to use the index in the query. The **present** is the corresponding predicate to the **atinstant** operator.

Tables 3 and 4 present the results of the temporal query for the various approaches in terms of number of disk accesses for the *U1lin* and *TrainsL* data sets, respectively. Both **present** and **atinstant** operators perform a binary search in the movement. In the *U1lin* data set, since we have 8,594 pages per tuple in average we could expect to have 13 reads per tuple, which is $\log_2(8594)$, which gives us 169 reads for the query without using an index. We achieve numbers similar to this analytical expectation.

When using standard indexing approaches, only for the queries with lower selectivies, the number of disk accesses is reduced. For the times in the middle of the data set, all tuples return in the query. However, with double indexing, we could reduce considerably the number of disk accesses also for the non-selective queries, given the fact that the selectivity inside the complex object is high, which means that the size of the returned object is considerably smaller than the original one.

Table 3: Performance of the temporal queries for the U1lin data set.

Selectivity (# of tuples returned)	1	13	13	13	1
	Disk accesses				
u1lin	173	173	167	168	180
u1lin_TripL_dT_eObj_iStd	13	174	168	169	14
u1lin_TripL_dT_eUnt_iStd	19	179	175	187	20
u1lin_TripL_dT_eUnt_iDbl	7	19	21	33	7

For the *TrainsL* data set we have a higher selectivity even for the query time instants in the middle of the data set. In this case, even the traditional indexing approach reduces considerably the number of disk accesses. With the double indexing approach we could reduce the number of disk access by a factor close to 2.

Table 4: Performance of the temporal queries for the TrainsL data set.

Selectivity (# of tuples returned)	100	2,250	2,400	2,400	50
	Disk accesses				
trainsl	106,787	118,232	131,333	144,286	155,978
trainsl_TripL_dT_eObj_iStd	108	4,237	4,646	4,593	154
trainsl_TripL_dT_eUnt_iStd	110	4,246	4,644	4,598	156
trainsl_TripL_dT_eUnt_iDbl	110	2,412	2,568	2,566	57

4.2 Spatial queries

For spatial queries we used the **at** operator receiving a moving point and a point p as argument. We performed the following query with different values of p, which correspond to some positions at the five values of i chosen in the last sub-section.

```
select val(trip at p) as pos
   from trains
   where trip passes p;
```

As done before, we added the **passes** operator in the *where* clause, since it is the corresponding predicate for the **at** operator.

Tables 5 and 6 present the results of the spatial query for the various approaches in terms of number of disk accesses for the *U1lin* and *TrainsL* data sets, respectively.

Table 5: Performance of the spatial queries for the U1lin data set.

Selectivity (# of tuples returned)	13	13	13	13	13
	Disk accesses				
u1lin	110,466	110,466	110,466	110,466	110,466
u1lin_TripL_dS_eObj_iStd	110,467	110,467	110,467	110,467	110,467
u1lin_TripL_dS_eUnt_iStd	111,803	111,803	114,281	111,803	111,803
u1lin_TripL_dS_eUnt_iDbl	19,752	19,752	35,251	19,752	19,752

Table 6: Performance of the spatial queries for the TrainsL data set.

Selectivity (# of tuples returned)	505	290	290	290	400
	Disk accesses				
trainsl	203,192	203,192	203,192	203,192	203,192
trainsl_TripL_dS_eObj_iStd	4,682	3,176	3,176	3,176	2,008
trainsl_TripL_dS_eUnt_iStd	2,073	1,171	1,171	1,171	2,008
trainsl_TripL_dS_eUnt_iDbl	685	312	312	310	428

For the *U1lin* data set, all queries are non-selective and standard indexing techniques cannot improve the performance of the queries, just adding extra overhead in the query processing. This non-selectivity occurs because all the trains pass through the Mehringdamm Station lots of times during their journeys. The double index approach could reduce the number of disk accesses by a factor of 5, because only a portion of the movement must be read from disk.

For the *TrainsL* data set, the query is selective and the index approaches give a good improvement. But even in these cases, the double indexing is still perform at least 3 or 4 times better.

4.3 Spatio-temporal queries

For spatio-temporal queries we combine a spatial and a temporal predicate. Internally, a spatio-temporal index is used under such combination. We performed the following query with different values of p and i, corresponding to the ones from the two last sub-sections.

```
select val(trip atinstant i) as pos
  from trains
  where (val(trip atinstant i) = p)
```

Tables 7 and 8 present the results of the spatio-temporal query for the various approaches in terms of number of disk accesses for the *U1lin* and *TrainsL* data sets, respectively.

Table 7: Performance of the spatio-temporal queries for the U1lin data set.

Selectivity (# of tuples returned)	1	13	13	13	1
	Disk accesses				
u1lin	173	185	205	181	181
u1lin_TripL_dST_eObj_iStd	13	187	206	182	15
u1lin_TripL_dST_eUnt_iStd	19	198	146	114	21
u1lin_TripL_dST_eUnt_iDbl	9	26	112	20	9

For these queries performed in the *U1lin* data set, their selectivity in terms of number of tuples returned is the same as for the temporal queries and their selectivity in terms of the size of the moving objects' trajectories is similar. Therefore, these results are quite similar to the ones achieved for the temporal queries.

Table 8: Performance of the spatio-temporal queries for the TrainsL data set.

Selectivity (# of tuples returned)	1	10	12	12	2
	Disk accesses				
trainsl	106885	122051	135199	148258	156076
trainsl_TripL_dST_eObj_iStd	9	157	162	177	13
trainsl_TripL_dST_eUnt_iStd	9	11	11	12	13
trainsl_TripL_dST_eUnt_iDbl	9	9	9	9	9

Given the high selectivity of these queries in the *TrainsL* data set, the standard index approaches are enough, reducing considerably the number of disk accesses. However, the double indexing approach still gives an improvement in performance of 20% in some cases.

It is important to note that the double indexing approach was never worse than the standard indexing approaches.

5 Conclusions

In this paper, we aimed to show the need for the multiple entry and the double indexing approaches. The operators in Secondo that allow one to create and query such indexes are presented. Following the presentation of the operators, examples on how to use them are also provided. Finally, an experimental evaluation of the proposed approaches is done in order to show their improvements in terms of efficiency with some spatial, temporal, and spatio-temporal queries. The double indexing approach showed the best overall performance in this comparison. Moreover, in our experiments the double indexing approach was never worse than any of the other ones.

As future work, we plan to provide optimization rules that take into consideration not only the selectivity in terms of the number of tuples, but also the selectivity in terms of the tuple size. These rules would enable the optimizer to better decide when to use the multiple entry and the double indexes present in the database.

References

[1] T. Behr and R. H. Güting. Fuzzy spatial objects: An algebra implementation in Secondo. In *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE)*, pages 1137–1139, 2005.

[2] V. T. de Almeida, R. H. Güting, and T. Behr. Querying moving objects in Secondo. In *Proc. of the 7th. Intl. Conf. on Mobile Data Management (MDM)*, page 47, 2006.

[3] S. Dieker and R. H. Güting. Plug and play with query algebras: Secondo – a generic DBMS development environment. In *Proc. of the Intl. Database Engineering and Applications Symposium (IDEAS)*, pages 380–392, 2000.

[4] S. Dieker, R. H. Güting, and M. R. Luaces. A tool for nesting and clustering large objects. In *Proc. of the 12th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 169–181, 2000.

[5] R. H. Güting, T. Behr, V. T. Almeida, Z. Ding, F. Hoffmann, and M. Spiekermann. Secondo: An extensible DBMS architecture and prototype. Technical Report 313, Fernuniversität Hagen, Fachbereich Informatik, 2004.

[6] R. H. Güting, V. T. de Almeida, D. Ansorge, T. Behr, Z. Ding, T. Höse, F. Hoffmann, M. Spiekermann, and U. Telle. Secondo: An extensible DBMS platform for research prototyping and teaching. In *Proc. of the 21st Intl. Conf. on Data Engineering (ICDE)*, pages 1115–1116, 2005.

[7] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *VLDB Journal*, 15(2):143–164, 2006.

[8] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *Proc. of 26th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 934–945, 2005.

[9] P. Seshadri and A. N. Swami. Generalized partial indexes. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 420–427, 1995.

[10] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.