Evaluation of indexing strategies for possibilistic queries based on indexing techniques available in traditional RDBMS

Juan Miguel Medina[a,*], Carlos D. Barranco[b], Olga Pons[a]

[a]*Department of Computer Science and Artificial Intelligence, University of Granada C/ Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain.*
[b]*Division of Computer Science, School of Engineering, Pablo de Olavide University Ctra. Utrera km. 1, 41013 Seville, Spain.*

Abstract

A common way to implement a fuzzy database is on top of a classical Relational Database Management Systems (RDBMS). Given that almost all RDBMS provide indexing mechanisms to enhance classical query processing performance, finding ways to use these mechanisms to enhance the performance of flexible query processing is of enormous interest.

This work proposes and evaluates a set of indexing strategies, implemented exclusively on top of classical RDBMS indexing structures, designed to improve flexible query processing performance, focusing in the case of possibilities queries. Results show the best indexing strategies for different data a query scenarios, offering effective ways to implement fuzzy data indexes on top of a classical RDBMS.

Keywords: Possibilistics query, Relational databases, Fuzzy databases, Database indexing, Fuzzy database indexing.

1. Introduction

Fuzzy set theory is a paradigm shift which helps to solve classical, and non-classical, problems in a more convenient way than crisp systems by softening set boundaries. The database world has taken advantage of fuzzy set theory by using it as a way to manage imprecise, uncertain and inapplicable data (called fuzzy data in this framework) and to model and process flexible queries. As a result, a significant number of proposals on fuzzy database models Fukami et al. (1979); Umano (1982); Prade & Testemale (1984); Testemale (1986); Medina et al. (1994); Caluwe (1997) and flexible querying Bosc & Pivert (1995) have been published. There have also been proposed various implementations of fuzzy database management systems (FDBMS) Galindo et al. (1998); Kacprzyk & Zadrożny (1995), in which imprecise, uncertain and inapplicable data can be managed and/or flexible queries can be processed.

*Corresponding author

Email addresses: medina@decsai.ugr.es (Juan Miguel Medina), cbarranco@upo.es (Carlos D. Barranco), opc@decsai.ugr.es (Olga Pons)

The emergence of fuzzy databases means that a new tool is available for developing novel applications which would process very large data sets pervaded with imprecision and vagueness using fuzzy methods. As these novel applications prove their potential as prototypes, they are integrated into real-world environments. In this kind of environment, high performance, scalability and availability are required for applications, and subsequently for each of their components, particularly for their underlying FDBMS.

Many authors claim that FDBMS should not be developed from scratch, as building an efficient database engine is a costly task, but on top of an existing DBMS. In fact, this approach makes possible to take advantage of the qualities of the underlying crisp DBMS. One solution following this idea has been proposed in Barranco et al. (2008c) where a model of fuzzy object-relational DBMS (FORDBMS) is described. This proposal takes advantage of the extension mechanisms of a recent commercial object-relational DBMS (ORDBMS) in order to build user-defined data types to seamlessly represent, store, query and manage fuzzy data by modeling them as native database objects.

In addition to an efficient database engine, indexing has always been a key technique to improve performance in database systems. Considering today's world challenges, as Big Data, indexing becomes a must. This is also true for the case of fuzzy databases and of critical importance for their applicability, specially those indexing techniques that can be built of top of a classical DBMS.

This paper studies, adapts and evaluates the most relevant indexing techniques proposals,and propose and evaluate some novel ones, for enhancing the performance of flexible queries, based on a possibilistic measure (c.f. Sect. 2), that can be implemented on top of a classical RDBMS. This last premise implies not considering indexing techniques devised without the implementation on top of an RDBMS in mind (i.e. specific indexes not available in a classical RDBMS).

The paper is organized as follows. The concept of fuzzy numerical data, the possibilistic queries and the principles of indexing for such kind of queries are described in Section 2. Section 3 describes the main elements available in classical RDBMS for query optimization. Section 4 briefly introduces related work on fuzzy data indexing. Section 5 describes all indexing techniques for possibilistic queries which are studied in this paper. Section 6 presents the experimental methods and analyzes the the results. Finally, Section 7 contains the concluding remarks and explores future lines of research.

2. Fuzzy data and possibilistic queries

This paper focuses on the search of the best performing indexing methods for fuzzy numerical when querying using a possibility measured atomic flexible condition. This section explains the main concepts involved in this task.

2.1. Fuzzy numerical data

For the purposes of the paper, a *fuzzy numerical value* is an imprecise and/or vague numerical value. This kind of values are stored in a fuzzy database when the precise value of a numerical attribute is not known. Examples of this kind of data for an age attribute are "about five" or "between 20 and 30". A value of this kind of data is modeled by a convex possibility distribution defined on an underlying domain D in which a linear

order relation is defined. The function that characterises the possibility distribution maps every element $d \in D$ to the possibility degree of the fact that d is the actual value of the attribute.

When stored in a fuzzy database, *fuzzy numerical values* are usually represented by a trapezoidal possibility distribution $[\alpha, \beta, \gamma, \delta]$, whose membership function is defined as shown in Eq. 1. For this reason, this kind of data is generically referred as trapezoids in the rest of the paper.

$$\mu_{[\alpha,\beta,\gamma,\delta]}(x) = \begin{cases} 1 & \text{if } \beta \leq x \leq \gamma \\ \frac{x-\alpha}{\beta-\alpha} & \text{if } \alpha < x < \beta \\ \frac{\delta-x}{\delta-\gamma} & \text{if } \gamma < x < \delta \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

A trapezoidal distribution can also represent crisp values, as crisp intervals $[a, b]$ (in the form $[a, a, b, b]$) and crisp values n as $[n, n, n, n]$.

These trapezoidal distributions can be stored in a database as four numerical attributes. Therefore, a relational table able to store fuzzy numerical data can be created through a SQL sentence like this:

```
CREATE TABLE Fuzzy_tab (alpha number,beta number,gamma number,delta number)
```
(2)

2.2. Flexible conditions

A flexible condition is a gradual restriction imposed on the values of an attribute. This restriction is modeled as a fuzzy set of acceptable values which must be characterized by a convex membership function. Some instances of this kind of conditions imposed on an age attribute are "around 30" and "in his/her 20s".

Applying a flexible condition on a set of rows, results on tow fuzzy subsets of these rows: one containing the row *possibly* satisfying the flexible condition, and one formed by the rows *necessarily* satisfying the condition (which is not covered for the purposes of this paper, as it requires specially designed indexing methods). When we apply a flexible condition to get the first fuzzy subset of rows, we refer to the condition as a possibility measured flexible condition. The membership function of the fuzzy set of rows that possibly satisfy the fuzzy condition C on the attribute A is defined in the Eq. 3.

$$\Pi(C/r) = \sup_{d \in D(A)} \big(\Pi_{A(r)}(d), \mu_C(d)\big) \tag{3}$$

where, $D(A)$ is the underlying domain associated to the fuzzy attribute A, $\Pi_{A(r)}$ is the possibility distribution which describes the fuzzy value of the attribute A for the row r, and μ_C is the membership function defining the fuzzy condition C.

A flexible condition is typically used in conjunction with a crisp relational comparator for setting a threshold (i.e. a minimum) for its fulfilment degree. The typical expression for applying a threshold T is $\Pi(C, r) \geq T$, except when the threshold is 0. In the latter case, the expression $\Pi(C, r) > T$ is applied. This combination is called an atomic flexible condition for the purposes of this paper and is notated as $\langle C, T \rangle$.

2.3. Indexing principle

The fundamentals of all indexing strategies evaluated in this work are the pre–selection criterion. This criterion lets us, from the index, to pre–select the rows of a table that could satisfy an atomic flexible condition for a given threshold or, in other words, remove from the result set the rows that surely not satisfy a given atomic flexible condition. This results in the pre–selection set. Of course, after determining this set, the selection set (the set of rows satisfying the given atomic flexible condition) is determined filtering the false positives at the pre–selection set.

This criterion, introduced in Bosc & Galibourg (1989), when applied on trapezoidal fuzzy attribute values $\Pi_{A(r)} = [\alpha_{A(r)}, \beta_{A(r)}, \gamma_{A(r)}, \delta_{A(r)}]$ and $l_{base(\langle C,T\rangle)}$ to find those rows r that could satisfy $\langle C, T\rangle$, the pre–selection criterion is expressed as Eq. 4 shows, where, $l_{base(\langle C,T\rangle)}$ and $u_{base(\langle C,T\rangle)}$ are, respectively, the infimum and the supremum of $base(\langle C, T\rangle)$, defined as in Eq. 5.

$$ps'(C/r, T) \Longleftrightarrow \delta_{A(r)} \geq l_{base(\langle C,T\rangle)} \wedge \alpha_{A(r)} \leq u_{base(\langle C,T\rangle)}. \tag{4}$$

$$base(\langle C, T\rangle) = \begin{cases} supp(C), & T = 0 \\ supp_T(C), & 0 < T \leq 1 \end{cases} \tag{5}$$

Note that, the proposed pre–selection criterion slightly increases the occurrence of false positives in some extreme cases when $\delta_{A(r)} = l_{base(\langle C,T\rangle)}$ or $\alpha_{A(r)} = u_{base(\langle C,T\rangle)}$ if these values are not part of the support of the trapezoidal distributions $A(r)$ and C. For sake of simplicity, from now on, we will note $l_{base(\langle C,T\rangle)}$ and $u_{base(\langle C,T\rangle)}$ as L_{CT} and U_{CT}, respectively.

The Fig. 1 illustrates the computation of the pre–selection and the selection subsets for a set of two rows r, s on which we apply a flexible condition $\langle C, T\rangle$ on the values of attribute A, with $\mu_C = [\alpha, \beta, \gamma, \delta]$, $\Pi_{A(r)} = [\alpha_{A(r)}, \beta_{A(r)}, \gamma_{A(r)}, \delta_{A(r)}]$ and $\Pi_{A(s)} = [\alpha_{A(s)}, \beta_{A(s)}, \gamma_{A(s)}, \delta_{A(s)}]$.

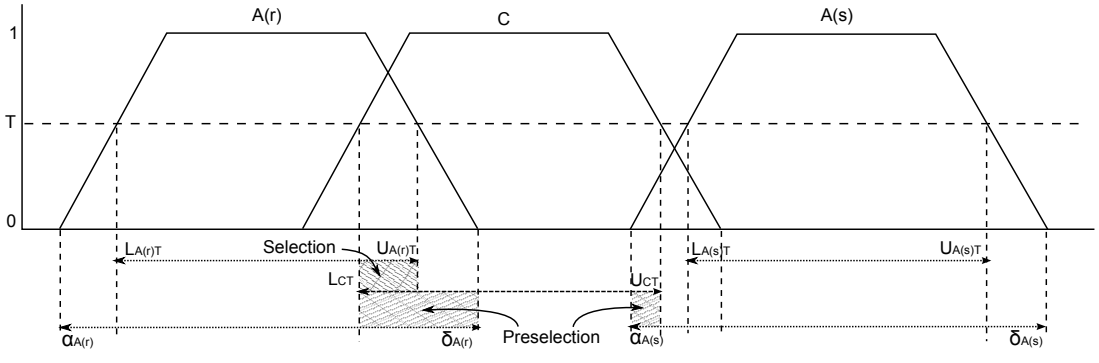


Figure 1: Example of the computation of the pre–selection and the selection sets.

It is straightforward to visually check that $A(r)$ and $A(s)$ satisfy the pre–selection criterion because their respective supports overlap the interval $[L_{CT}, U_{CT}]$, whose limits

are calculated by means of the next expressions:

$$L_{CT} = T * \beta + (1 - T) * \alpha \tag{6}$$
$$U_{CT} = T * \gamma + (1 - T) * \delta$$

To evaluate if a pre–selected row, $A(t)$, satisfies the selection criterion for the given threshold T, it is required that it fulfils the next condition of selection:

$$L_{A(t)T} \leq U_{CT} \wedge U_{A(t)T} \geq L_{CT} \tag{7}$$

where, $L_{A(t)T}$ and $U_{A(t)T}$ values are computed as follows:

$$L_{A(t)T} = T * \beta_{A(t)} + (1 - T) * \alpha_{A(t)} \tag{8}$$
$$U_{A(t)T} = T * \gamma_{A(t)} * T + (1 - T) * \delta_{A(t)}$$

Notice that, in the example of Fig 1, the trapezoid $A(r)$ satisfies the selection condition given in Eq. 7, whilst the trapezoid $A(s)$ does not.

2.4. SQL expressions for queries involving possibilistic conditions

If we represent trapezoidal data in a table as in Expr. 2, if the trapezoidal representation for C is $[alpha, beta, gamma, delta]$ and if the threshold is T, then, taking into account the Eq. 4, we can express the SQL statement for computing the pre–selection step as follows:

$$\texttt{SELECT * FROM Fuzzy\_tab WHERE alpha} \leq U_{CT} \texttt{ AND delta} \geq L_{CT}\texttt{;} \tag{9}$$

where L_{CT} and U_{CT} are calculated as Eq. 6 shows.

Likewise, based on the Eqs. 7 and 8, the selection condition can be expressed in SQL by means of the following statement:

$$\begin{aligned}\texttt{SELECT * FROM Fuzzy\_tab WHERE (T*beta + (1-T)*alpha)} &\leq U_{CT} \\ \texttt{AND (T*gamma+ (1-T)*delta)} &\geq L_{CT}\texttt{;}\end{aligned} \tag{10}$$

Note that the predicate values of the sentence 10 are only known at execution time, making impossible to use an indexing technique to speed up its execution. On other hand, the sentence 9 is a good candidate to profit from indexing on fields `alpha` and `delta` fields values. Actually, most of the considered indexing techniques pursue an efficient way to storage alpha and delta values in order to speed up the retrieval of the tuples that satisfy the pre–selection condition. Interestingly, if an indexing strategy allows to store in the index all values used in the selection condition, the selection step could be performed on the index entries, without the need to retrieve the values on the table.

3. Query optimization in Traditional RDBMS

When a SQL query is sent to an RDBMS, one expects, not only to get the correct results but that the response time is as shorter as possible. To reach this last goal, the RDBMS provide a system that processes the original query and generates an execution

plan that tries to minimize the execution time of the query. This system is called query optimizer (QO henceforward).

The input of the QO is a SQL statement and the output is an execution plan that describes the optimum method of execution. The plan shows the combination of steps the RDBMS uses to execute a SQL statement.

The QO considers many factors related to the objects and the conditions in the query to try to determine an optimal execution plan. The QO generates a set of potential plans for the SQL statement based on available access paths. Then, it estimates the cost of each plan based on statistics in the data dictionary, which include information on the data distribution and storage characteristics of the tables and indexes accessed by the statement. Finally, the QO chooses the plan with the lowest cost.

The QO does not always succeed choosing the optimal execution plan. For this reason, the RDBMS provides mechanisms for selecting one of the alternative execution plans, and also provides clauses (called *Hints*) to force the QO to use particular access paths or strategies. This feature will be of particular interest in our experiments, as we will use clauses to force the use of the indexing strategy that is being evaluated.

Given that our experimentation has been carried out using Oracle© RDBMS 11.2, in this section we will describe the optimization aspects relevant to our work available in this RDBMS. Further information can be found in Oracle© manuals, especially in Oracle (2012).

3.1. DBMS Indexing techniques

As the indexing techniques studied in this paper must be built on top of existing traditional RDBMS indexing mechanisms, in this section we review them.

The indexes the classical RDBMS available in can be categorized as follows:

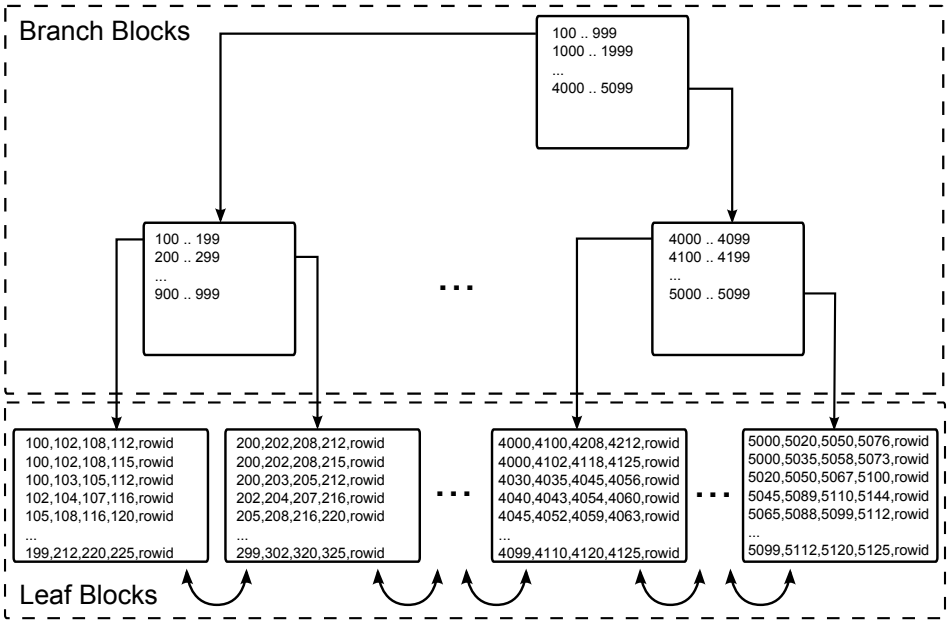


Figure 2: Structure of a composite B-tree index

B-tree indexes B-trees (balanced trees), are the most common type of database index. A B-tree index is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches. Our proposal uses composite indexes, which are B-tree indexes ordered by means of two or more attributes. Fig. 2 shows an example of the internal structure of a composite index defined on the attributes: alpha, beta, gamma and delta from a table containing fuzzy data. The first attribute (alpha) is the key of the index. The remaining attributes determine the partial order of those fuzzy data which have the same value for alpha.

The index is organized by means database blocks which are of two types: branch blocks and leaf blocks. The former are used to search values and the second to store values. Let's see by an example how this index is used to find a determined value. To search a tuple that contains the trapezoidal fuzzy value given by [200,202,208,215], the process starts searching in the root block the range that comprises the first key value (200); in this case, it is the 100..999 range, which points to the first branch block at the second level. Then, in this block the range that includes the key value (200..299) it is searched for and it points to the second leaf block. This block stores an ordered list of key values which is now sequentially explored to find the searched key, the second in our case; this element includes the pointer to the table row that contain the searched value.

The preceding operation is generally called an index scan. Some important considerations about index scan are: 1) if h is the height of the B-tree, to find a key value it is necessary h *I/O* operations. 2) If a SQL statement accesses only indexed columns, then the database reads values directly from the index rather than from the table.

There are several types of index scans, for our purposes the most important is Index range scan; it is applied when the predicates include a range condition on an attribute which constitutes the first value of the index key. The index is used to search one of the range bounds, then the leaf blocks are scanned (in ascending or descending order) to retrieve the rowids until the second bound is reached.

Key	**Rowid Range**	**Bitmap**
100	Rowid0 .. Rowid7	00100000
100	Rowid8 .. Rowid15	00000100
102	Rowid8 .. Rowid15	00001000
105	Rowid8 .. Rowid15	01000000
...	...	...
199	Rowid32 .. Rowid39	01000000

Table 1: Example of the structure and content for the first leaf block of a bitmap index.

Bitmap indexes In a bitmap index, the database stores a bitmap for each index key. In a conventional B-tree index, one index entry points to a single row. In a bitmap

index, each index key stores pointers to multiple rows. The database uses a B-tree index structure to store bitmaps for each indexed key.

If we create a bitmap index on the same table than the used in the previous example, using the alpha value of the trapezoidal representation as the key, the structure for the branch blocks is the same that the one shown in Fig. 2; however, the leaf nodes contain bitmap entries for each distinct key value with the following structure: key value, range of rowids, bitmap for rowids in the range.

The first leaf block in Fig. 2 shows three entries for the value of the key 100, one for the values 102, 105 and 199, among other entries not specified. According to this information, the content of this leaf block could be similar than the one shown in Table 1.

The information shown in Table 1 is interpreted as follows. The first entry in this table indicates that third row of the indexed database table contains the value 100 for the indexed attribute. The second entry informs that the fourteenth row of the database table also contains the 100 value. Likewise, the third, fourth and final entries indicate that the values 102, 105 and 109 are respectively the values for the indexed attribute at rows 13^{th}, 10^{th} and 34^{th}.

R*Tree Indexes An R-tree index is designed to index spatial data. In this work we consider the use of Oracle Spatial© feature, which uses a R-Tree to index spatial data of up to four dimensions. Because our proposal will work on a bidimesional space, we will illustrate the structure and query functioning of this index on such space. Fig. 3 shows the structure of a R-tree index after the insertion of the geometries 1 to 9 (which are shown shaded in the left side of the figure). An R-tree index approximates each geometry by a single rectangle that minimally encloses the geometry (called the minimum bounding rectangle, or MBR). The R-tree structure is composed by a hierarchy of MBR of the geometries in the layer where: a, b, c, and d are the leaf nodes of the R-tree index and contain MBR of geometries together with pointers to the geometries, A contains the MBR of a and b, and B contains the MBR of c and d, and finally, the *root* contains the MBR of A and B (that is the entire area shown).

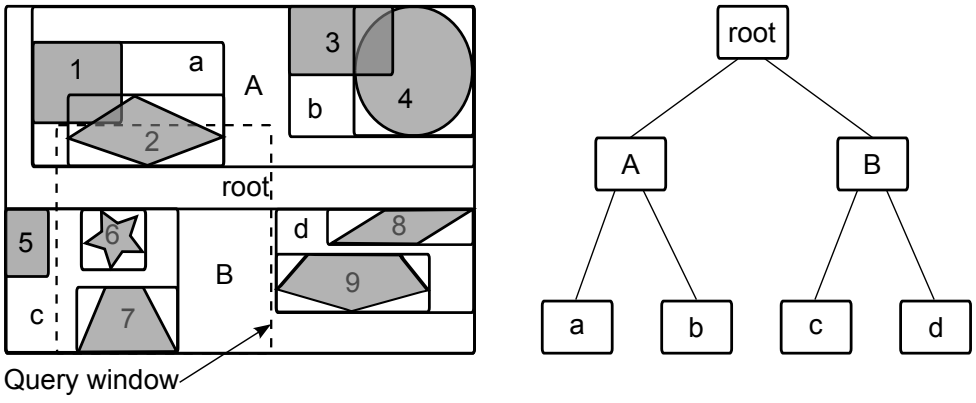


Figure 3: Structure and query on a R-tree index

The processing of a spatial query using this index is as follows. Firstly, the query is expressed by means of a *query window* as Fig. 3 (left) shows. Then, the index, Fig. 3 (right), is traversed from *root*, finding that the *query window* overlaps with the MBRs of nodes A and B. Finally, considering the node A, it checks which subnodes overlap the *query window*. In this case, the node a is the only one, which is subsequently retrieved to get the MBRs of the geometries it includes (1 and 2). The geometry 2 is then retrieved as a candidate result because is the only one whose MBR overlaps with the *query window*; once retrieved, the concrete spatial operator is evaluated on the geometry to evaluate if it will be on the final result of the query. Likewise, considering the node B, the index determines that the MBR of c is the only one that overlaps with the *query window*. Such node is retrieved and it is verified that the MBRs of the geometries 6 and 7 completely inside the *query window*; so these geometries are recovered as a result of the query.

Function-based indexes This type of index includes columns that are either transformed by a function, for instance the SQL UPPER function, or included in an expression. B-tree or bitmap indexes can be function-based.

Application domain indexes This type of index is created by a user for data in an application-specific domain. The physical index need not use a traditional index structure (although it can be used) and can be stored either in the Oracle© database as tables or externally as a file. This kind of index, combined with user defined data types (UDTs) allows to extend the capabilities of the RDBMS to handle new data types. These capabilities can be used to finally implement some of the indexing strategies proposed in this work.

3.2. Execution Plan

As we have mentioned, when executing a query, the QO evaluates several execution plans and selects the optimal one. To examine which is the execution plan selected, the SQL statement `EXPLAIN PLAN FOR <sql_sentence>` is used, where the `<sql_sentence>` can be a query or any other kind of sql statement. As result of the execution of this statement, the execution plan for the considered sentence is stored into a table (by default plan_table). After that, this table can be queried to get the execution plan for the analyzed statement. To get a formatted output of an execution plan from the plan_table, we can use a predefined function `dbms_xplan.display()`. The following example illustrates the process to get the execution plan for a query which implements a pre–selection condition using the expression in Eq. 9:

```
EXPLAIN PLAN FOR SELECT count(*) FROM TAB WHERE (alpha <=52188 AND delta >=51936.5);
```

then, we execute:

```
SELECT * FROM table(dbms_xplan.display());
```

as result we get:

```
PLAN_TABLE_OUTPUT
----------------------
Plan hash value: 1291143156
-----------------------------------
| Id  | Operation          | Name  |
-----------------------------------
|   0 | SELECT STATEMENT   |       |
|   1 |  SORT AGGREGATE    |       |
|*  2 |   TABLE ACCESS FULL| TAB   |
-----------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
2 - filter("DELTA">=51936.5 AND "ALPHA"<=52188)
```

The execution plan shown is interpreted in this way:

1. The DBMS performs a TABLE ACCESS FULL on the table Tab; this operation means that the RDBMS traverses all tuples of this table for returning the ROWID of those that satisfy the "filter" predicate.
2. The next step is a SORT AGGREGATE operation which retrieves a single row as the result of applying a group function to a group of selected rows (count(*)).
3. The higher level operation is SELECT STATEMENT, which indicates that the analyzed statement is a SELECT statement.

Each operation implies a step in the query processing with the format: `<Operation> <Option> <Object>`. Furthermore, the operation can be based on the satisfaction of a predicate which is described in the "Predicate Information" section of the output. They can appear two kinds of predicates: 1) *access predicates* used to locate rows in an access structure. For example, start or stop predicates for an index range scan, and 2) *filter predicates*, to filter rows before returning them to the user, *filter predicates* are only applied during the leaf node traversal, they do not contribute to the start and stop conditions and do not narrow the scanned range.

3.2.1. Execution plan options

Next, we will detail the operations and options more relevant to our work:

- INDEX. Indicates the use of a B-tree index as access method (the index shown in the Name column). The option UNIQUE SCAN shows that the retrieval will be of a single rowid from the index. The option RANGE SCAN informs us that the retrieval will be of one or more rowids in ascending order.

- BITMAP INDEX. Indicates that the Bitmap index shown in the Name column will be used. The option RANGE SCAN indicates that bitmaps for a key value range will be retrieved

- BITMAP. With the option MERGE indicates the merging of several bitmaps resulting from a range scan into one bitmap. With the option AND, it informs us of the application of a bitwise AND of these bitmaps. The option CONVERSION TO ROWIDS converts bitmap representations to current rowids that can be used to access the table.

- DOMAIN INDEX. Indicates the retrieval of one or more rowids from a domain index (the index shown in the Name column).
- INTERSECTION is an operation that accepts two sets of rows and returns the intersection of the sets. The UNION-ALL option indicates the application of an operation which accepts two sets of rows and returns the union of the sets, without eliminating duplicates.
- NESTED LOOPS. Operation that accepts two sets of rows, an outer set and an inner set; the database compares each row of the outer set with each row of the inner set, returning rows that satisfy a condition.
- TABLE ACCESS. With the option BY INDEX ROWID, it establishes the retrieval of the tuples of the indicated table by means of the use of the rowids returned by the previous index operation.
- SORT. With the option AGGREGATE, it indicates the retrieval of a single row that is the result of applying a group function to a group of selected rows. The UNIQUE option establishes an operation sorting a set of rows to eliminate duplicates.
- VIEW. Indicates an operation performing a view query and then, returning the resulting rows back to another operation.

3.2.2. Hints

As mentioned before, the QO evaluates and tries to select the best execution plan, but if we want to force the use of a specific access method, we can use the "HINT" clause that is enclosed by /*+ */ next to the SELECT clause in the way:

```
SELECT /*+ INDEX (TAB BT_DABG)*/ count(*) FROM TAB WHERE (alpha <=52188 AND delta >=51936.5)
AND (beta*.5 + .5* alpha <=52188 AND  gamma*.5+ .5* delta >=51936.5);
```

The resulting execution plan is:

```
PLAN_TABLE_OUTPUT
--------------------
Plan hash value: 3749191139
-------------------------------------
| Id  | Operation         | Name    |
-------------------------------------
|   0 | SELECT STATEMENT  |         |
|   1 |  SORT AGGREGATE   |         |
|*  2 |   INDEX RANGE SCAN| BT_DABG |
-------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
 2 - access("DELTA">=51936.5 AND "DELTA" IS NOT NULL)
     filter("ALPHA"<=52188 AND "BETA"*.5+.5*"ALPHA"<=52188 AND "GAMMA"*.5+.5*"DELTA">=51936.5)
```

In this example, the hint `/*+ INDEX (TAB  BT_DABG)*/` forces the QO to perform an `INDEX RANGE SCAN` on the index `BT_DABG`, which is a B-tree index defined on the attributes delta, alpha, beta and gamma, accessing only the index entries that accomplish "access" predicate and then applying on them the "filter" condition (note that the last operation can be done on the index entries, because these entries store all the needed values). There are many available hints, but we will use one more, the `INDEX COMBINE` hint, which forces the QO to consider a list of indexes on evaluating the execution plan (see Section 5.2).

4. Related Work

The aim of this work is to determine the best indexing techniques for possibilistic queries that can be implemented using the resources that an RDBMS provides. This premise determines two key aspects: (a) in traditional RDBMS, the most determinant factor for performance in retrieval is the amount of retrieved database blocks, and (b) to be implemented into traditional RDBMS, an indexing technique only should require structures already available in traditional RDBMS.

Taking into account these considerations, we will review the most relevant proposals in the literature. Because the problem of enhancing possibilistic queries performance is based on the problem of enhancing the retrieval of intervals that intersect to a given one, we will also review indexing proposals oriented to the interval retrieval.

4.1. Indexing proposals for possibilistic queries

In the field of fuzzy databases there have been some research efforts aimed to include indexing mechanisms specially designed to handle flexible queries and imperfect data.

Among the literature, we can highlight the seminal paper Bosc & Galibourg (1989), as commented in sect. 2.3, where the principles for indexing in fuzzy database are laid. On top of this, there are proposals to extend traditional database indexes, as B^+-trees Barranco et al. (2008a, 2009), bitmaps Barranco & Helmer (2012) or multidimensional indexes as g-tree Liu et al. (1996), to make them able to support flexible querying on fuzzy data. Apart from this, there are proposals in the literature not using the above indexing principle, as some specially designed structures for fuzzy object-oriented databases Yazici et al. (2008).

On the set of the above proposals, only some of them are able to assist on open (i.e. non-fixed) query processing. Particularly, in the case of possibilistic open queries on numerical data, only some of the above proposals Barranco et al. (2008a) are suitable for the task.

4.2. Structures for indexing intervals

The most interesting proposal for the indexing of intervals is RI-tree, which was presented in Kriegel et al. (2000). RI-tree is a relational storage structure based on the main memory interval tree, Edelsbrunner (1983), and can be built on top of the SQL layer of any RDBMS. The structure of an RI-tree consists of a binary tree of height h which makes the range $[0..2^h - 1]$ of potential interval bounds accessible. It is called the virtual backbone of the RI-tree since it is not materialized but only the root value 2^{h-1} is stored in a metadata table. Traversals of the virtual backbone are performed purely arithmetically, by starting at the root value and proceeding in positive or negative steps of decreasing length 2^{h-1}, thus reaching any desired value of the data space in $O(h)$ CPU time and without causing any I/O operation. Upon insertion, an interval is registered at the highest node that is contained in the interval. For the relational storage of intervals, the value of that node is used as an artificial key.

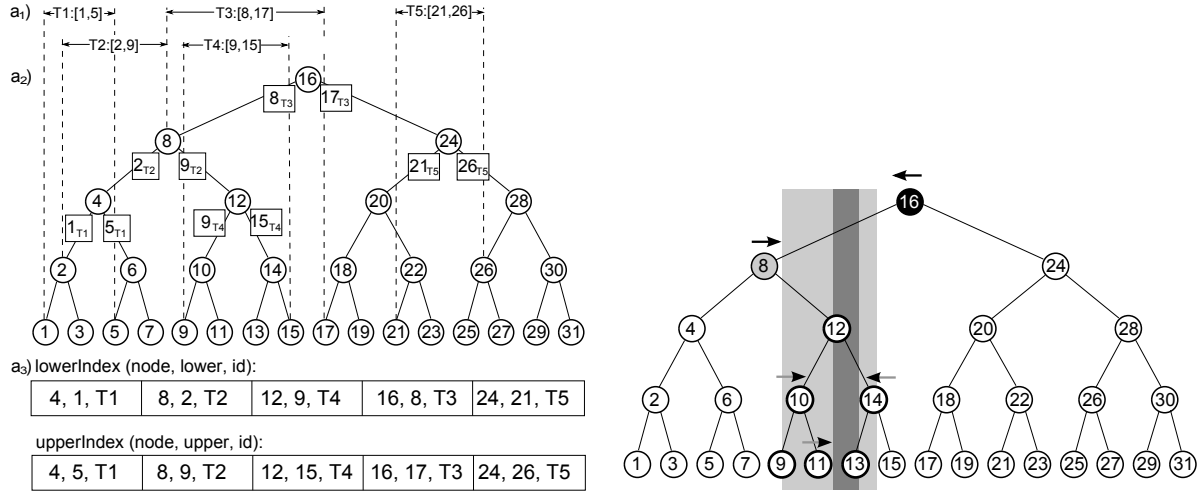
An instance of the RI-tree consists of two relational indexes, which in an extensible indexing environment, are preferably managed as index-organized tables. These indexes correspond to the relational schema *lowerIndex (node, lower, id)* and *upperIndex (node,*

upper, id) which store the artificial key value *node*, the *lower* and *upper* bounds, respectively, and the *id* of each interval. An interval is represented by a single entry in each of the two indexes, and therefore, *O(n/b)* disk blocks of size b suffice to store n intervals. For inserting or deleting intervals, the *node* values are determined arithmetically, and updating the indexes requires $O(\log_b n)$ *I/O* operations per interval.

Trap_id	**Alpha**	**Beta**	**Gamma**	**Delta**
Qry_trap	9	12	13	14
T1	1	2	3	5
T2	2	4	7	9
T3	8	12	14	17
T4	9	10	11	15
T5	21	23	25	26

Table 2: Example table of trapezoidal values and a queried value.

To illustrate the creation of the index contents in the RI-tree structure, consider the trapezoids $T1..T5$ shown in Table 2. If we want to index the support of these trapezoids, $[alpha, delta]$, to apply the indexing principle described in Sect. 2.3, the process is as shown in Fig. 4 (a). As we can see, for the interval $T1$, the value of the *node* assigned is 4, because it is the highest node in the virtual backbone between its interval bounds. In the same way, for the interval $T2$ the *node* assigned is 8, and so on.



(a) RI-tree example

(b) Query preparation step

Figure 4: (a) RI-tree expample: a_1) Support of the five trapezoids example, a_2) Virtual backbone and registration position, and a_3) Contents of the indexes lowerIndex and upperIndex. (b) Query preparation step for the query interval [9,14] (shaded in light gray): *leftQueries* {8}; rightQueries {16}; innerQueries {9-14}

The processing of interval intersection query $[start, end]$, based on the RI-Tree is performed in two phases: the procedural query preparation phase and the declarative query processing phase. To illustrate the process, see we will analyse the procedure by an example using as query interval the support of the *Qry_trap* described in Tab 2, that

is $[9, 14]$ (See Fig. 4 (b)). In the first phase, the virtual backbone is traversed descending from the *root* node down to *star* (9 in this example) and to *end* (14), respectively. The traversal is performed arithmetically without causing any I/O operations, and the visited nodes are collected in two different main-memory tables *leftQueries* and *rightQueries* both obeying the unary relational schema (*node*). Notice that *leftQueries* only collects the intermediate nodes adjacent by the left to *start* value (8 in this example) discarding the other nodes, because the upper value of the intervals that they represent is less than the *start* value. Respectively, *rightQueries* only collects the intermediate nodes adjacent by the right to *end* value, 14. The intervals with node value in *leftQueries* intersect the queried interval if its *upper* value is greater than or equal to *start*. In our example $T2$, with node value 8, satisfies this condition, so it intersects the interval $[9, 14]$.

Likewise, the intervals with node value in *rightQueries* intersect the queried interval if its *lower* value is less than or equal to *end*. $T3$ belongs to the node 16 and its *lower* value satisfies the last condition, so it intersects the interval $[9, 14]$. Whereas these nodes are taken from the paths, the set of all nodes between *start* and *end* belongs to the so-called *innerQuery*, which is represented by a single range query on the *node* values. All intervals registered at nodes from the *innerQueries*, ($T4$ in our example), are guaranteed to intersect the query, since they have at least their node value in common with the query interval and, therefore, will be reported without any further comparison of lower or upper values. The query preparation step is purely based on main memory and requires no I/O operations.

In the second phase, the transient tables are joined with the relational indexes *upperIndex* and *lowerIndex* by a single three-fold SQL statement:

```
SELECT id FROM upperIndex AS i JOIN :leftQueries USING (node) WHERE i.upper >= :start
UNION ALL SELECT id FROM lowerIndex AS i JOIN :rightQueries USING (node) WHERE i.lower <= :end
UNION ALL SELECT id FROM lowerIndex  // or upperIndex WHERE node BETWEEN :start AND :end
```

The *upper* point of each interval registered at nodes in *leftQueries* is compared to *start*, and the *lower* point of intervals in *rightQueries* is compared to *end*. The *innerQuery* corresponds to a simple range scan over the intervals with nodes in $[start, end]$. Because intervals are organized by the node value, the significant intervals which intersect the query interval are stored in contiguous ranges on disk. For a tree height of h, there are at most $2 \cdot h$ different ranges which have to be considered when processing the query interval. Since the output from the relational indexes is fully blocked for each join partner, the SQL query requires $O(h \cdot log_b n + r)$ I/Os to report r results from an RI-tree with n stored intervals (block size b).

5. Indexing techniques for possibilistic queries

In this Section we will describe several proposals based on the use of B-trees, RI-trees, Bitmap and R-trees (Oracle Spatial©) indexes. For each evaluated index strategy, we briefly describe its fundamentals, the necessary structure to implement it into an RDBMS, the rewritten query to force the use the defined indexes and the execution plan the host RDBMS generates to execute the query.

It must be noted that, in this section and in the performance evaluation section, we only use statements that retrieve amount of tuples satisfying the fuzzy condition, not the

set of such tuples. This decision allows to better focus the study in the performance of each strategy, avoiding considerations of how rows must be retrieved after index processing.

5.1. B-Tree based indexing

There are few approaches that use B-trees to enhance possibilistic queries performance. This section reviews existing and proposes some novel ones.

5.1.1. 2BPT indexing

In Barranco et al. (2008b) the authors propose an indexing technique to enhance possibilistic queries by means of the use of two indexes defined on alpha and delta, respectively. To pre–select the tuples that satisfy the query, it uses the first index to get the tuples that accomplish the condition alpha $\leq$ U_CT and uses the second one to get the tuples that satisfy the condition delta $\geq$ L_CT; then, the intersection of these sets is calculated to get the tuples that satisfy the pre–selection criterion. Finally, on these tuples, the selection criterion is applied.

We have adapted this technique to enhance the selection performance by means of the inclusion of all trapezoidal values into the defined indexes, in this way:

```
CREATE INDEX BT_ADBG ON TAB (ALPHA,DELTA,BETA,GAMMA); CREATE INDEX BT_DABG ON TAB (DELTA,ALPHA,BETA,GAMMA);
```

The query that forces the RDBMS to execute the query according to this indexing strategy is:

```
SELECT COUNT(*) FROM ((SELECT /*+ INDEX (TAB BT_ADBG)*/ rowid FROM TAB WHERE (alpha<=U_CT)
 AND (beta*T + alpha*(1-T)<=U_CT AND  gamma*T+ delta*(1-T)>=L_CT))
INTERSECT (SELECT /*+ INDEX (TAB BT_DABG)*/ rowid FROM TAB WHERE (delta>=L_CT)
 AND (beta*T + alpha*(1-T)<=U_CT AND  gamma*T+ delta*(1-T)>=L_CT)));
```

The execution plan for an example of query using this indexing strategy is:

```
-----------------------------------------
| Id  | Operation            | Name     |
-----------------------------------------
|   0 | SELECT STATEMENT     |          |
|   1 |  SORT AGGREGATE      |          |
|   2 |   VIEW               |          |
|   3 |    INTERSECTION      |          |
|   4 |     SORT UNIQUE      |          |
|*  5 |      INDEX RANGE SCAN| BT_ADBG  |
|   6 |     SORT UNIQUE      |          |
|*  7 |      INDEX RANGE SCAN| BT_DABG  |
-----------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
5 - access("ALPHA"<=40179); filter("BETA"*0+1*"ALPHA"<=40179 AND "GAMMA"*0+1*"DELTA">=40129)
7 - access("DELTA">=40129 AND "DELTA" IS NOT NULL)
  - filter("BETA"*0+1*"ALPHA"<=40179 AND "GAMMA"*0+1*"DELTA">=40129)
```

As we can see, the intersection operation (3) invoves two sort operations (4 and 6) on each set of rows obtained from the respective index range scan operations (5 and 7). It must be noted that the computational cost of the intersection is high ($O(n \cdot \log n)$). For the evaluation of performance of the next Section, we will refer to this strategy with the label **2BPT**.

5.1.2. Strategies based on the use of a single B-Tree index

Because of the computational cost of the intersection operation, we have considered an alternative based on the use of a single B-tree defined on either alpha or delta. For example, we will use a compound B-tree index defined on the fields alpha, delta, beta, gamma. The query statement that force the use of this index would be:

```
SELECT /*+ INDEX (TAB BT_ADBG)*/ count(*) FROM TAB
WHERE (alpha<=U_CT AND delta>=L_CT) AND (beta*T + alpha*(1-T)<=U_CT AND gamma*T+ delta*(1-T)>=L_CT);
```

The execution plan generated for an example query based on this strategy is:

```
-------------------------------------
| Id  | Operation         | Name    |
-------------------------------------
|   0 | SELECT STATEMENT  |         |
|   1 |  SORT AGGREGATE   |         |
|*  2 |   INDEX RANGE SCAN| BT_ADBG |
-------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("DELTA">=16803.5 AND "ALPHA"<=16838.5)
       filter("BETA"*.3+.7*"ALPHA"<=16838.5 AND "GAMMA"*.3+.7*"DELTA">=16803.5 AND "DELTA">=16803.5)
```

It worth noticing that given that the first attribute indexed is alpha, the queries with a low value for U_CT (what implies small amount of rows with a value of alpha $\leq$ U_CT) are the most benefited by the use of this index (this is the case in this example). We name this strategy as **1BT**.

5.1.3. Strategies based on the use of two B-Tree indexes

To try to solve the previous problem, we are going to define another compound B-tree index on the attributes (delta,alpha,beta,gamma). The idea is to avoid the inefficient intersection operation of the **2BPT** strategy and provide a mechanism to force the QO to use one of these indexes depending of the values of the query. Therefore, when the condition alpha $\leq$ U_CT involves fewer rows than the condition delta $\geq$ L_CT, the QO will use the B-tree Index based on (alpha,delta,beta,gamma), otherwise it will use the B-tree index based on (delta,alpha,beta,gamma). To adopt the better decision, the QO needs some statistics about the real distribution of the rows in the index in function of its alpha and delta values. In our case, Oracle© 11.2 automatically collects statistics during index creation and rebuilding. We only need to instruct Oracle© to consider the two defined indexes in its query optimization process, using the INDEX_COMBINE optimization hint:

```
SELECT /*+ INDEX_COMBINE (TAB BT_ADBG BT_DABG)*/ count(*) FROM TAB
WHERE (alpha<=U_CT AND delta>=L_CT) AND (beta*T + alpha*(1-T)<=U_CT AND gamma*T+ delta*(1-T)>=L_CT);
```

For the previous query example, the execution plan is the same and the QO uses the index BT_ADBG. The following execution plan is elaborated by the QO for a query that has a greater value for L_CT, and so the use of the index BT_DABG is more efficient.

```
-------------------------------------
| Id  | Operation         | Name    |
-------------------------------------
|   0 | SELECT STATEMENT  |         |
|   1 |  SORT AGGREGATE   |         |
|*  2 |   INDEX RANGE SCAN| BT_DABG |
-------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
2 - access("DELTA">=72659.1 AND "DELTA" IS NOT NULL)
    filter("ALPHA"<=72711.8 AND "BETA"*.9+.1*"ALPHA"<=72711.8 AND "GAMMA"*.9+.1*"DELTA">=72659.1)
```

We will refer to the above indexing strategy as **2ABT**.

5.1.4. RI-Tree based indexing

This indexing structure, described in Section 4.2, is designed to enhance the retrieval of intervals. Here, we will adapt it to better functioning on a traditional RDBMS and to operate on posibilistic queries. Again, to get better performance for possibilistic queries, we include the values of the attributes alpha, beta, gamma and delta into the indexes, to perform the selection step into the index.

There is a subtle difference when adapting the RI-tree structure to handle queries on trapezoidal data since we apply the indexing on the support of the trapezoids; this means that the result set depends on the threshold of the query. For example, the trapezoid $T4$ in the database example shown in Table 2, was assigned the *node* value 12, calculated from its support, in the *RI-tree* index (Fig. 4 (a)). When we use the trapezoid *Qry_trap* (Table 2) to perform a query, establishing a 0 threshold, the Fig. 4 (b) shows that the $T4$ *node* belongs to the *innerQueries* (light gray rectangle), and so, it is rightly retrieved in the result set. However, when a threshold of 1 is set, the lists *leftQueries*, *rightQueries* and *innerQueries* change containing, respectively, the nodes $\{8, 10, 11\}$, $\{14, 16\}$ and, $\{12 - 13\}$. Again, the indexed trapezoid belongs to the new *innerQueries* list, and following the original *RI-tree* algorithm, this trapezoid would be recovered; however, for this threshold, $T4$ becomes the interval $[10, 11]$ which does not intersect the queried interval $[12, 13]$ (dark gray rectangle). For this reason, for possibilistics queries, the indexed nodes which belong to the inner Queries list, do not necessarily satisfy the query, and it is necessary that they also satisfy the selection condition.

Our adaptation of the RI-tree technique to handle possibilistic queries is called possibilistic Ri-tree (**PRI** for short) and consists of the following structures:

- A *PL/SQL* implementation of the function *forkNode*, that generates a *node* value for the support of the fuzzy value (*[alpha,delta]*).
- A table (*Trape(fnode,alpha,beta,gamma,delta)*) that stores the trapezoidal representation of the fuzzy values together with the value returned by the *forkNode* function applied to the interval *[alpha,delta]* (*fnode*).
- We don't use the auxiliary tables *lowerIndex* and *upperIndex* of the original strategy, but we use two B-Tree indexes defined on the base table: one (*RI_nadbg*) built on the *fnode, alpha, delta, beta, gamma* attributes and, the other (*RI_ndabg*) on the *fnode, delta, alpha, beta, gamma* attributes. These indexes are automatically updated when the base table is updated, without the need of triggers for this task.
- Two *PL/SQL* functions *leftNodes* and *rightNodes*, which implement the transient tables *leftQueries* and *rightQueries* described in Section 4.2, respectively.

To perform a query according to this implementation, the *SQL* query must be rewritten as follows:

```
SELECT /*+ INDEX (Trape RI_ndabg)*/ count(*) FROM Trape
WHERE (fnode in (SELECT /*+ CARDINALITY(l,20) */ l.column_value FROM table(LeftNodes(L_CT)) l) --leftNodes
  AND delta >=L_CT) AND (beta*T + alpha*(1-T)<=U_CT AND gamma*T+ delta*(1-T)>=L_CT)
UNION ALL SELECT /*+ INDEX (Trape RI_nadbg)*/ count(*) FROM Trape
WHERE (fnode in (SELECT /*+ CARDINALITY(r,20) */ r.column_value FROM table(RightNodes(U_CT)) r) --rightNodes
  AND alpha <=U_CT) AND (beta*T + alpha*(1-T)<=U_CT AND gamma*T+ delta*(1-T)>=L_CT)
UNION ALL SELECT /*+ INDEX (Trape RI_nadbg)*/ count(*) FROM Trape
  WHERE (fnode BETWEEN L_CT AND U_CT) AND (beta*T + alpha*(1-T)<=U_CT AND gamma*T+ delta*(1-T)>=L_CT)
```

The execution plan for an example query using this strategy is:

```
-------------------------------------------------------
| Id  | Operation                          | Name       |
-------------------------------------------------------
|   0 | SELECT STATEMENT                   |            |
|   1 |  UNION-ALL                         |            |
|   2 |   SORT AGGREGATE                   |            |
|   3 |    NESTED LOOPS                    |            |
|   4 |     SORT UNIQUE                    |            |
|   5 |      COLLECTION ITERATOR PICKLER FETCH| LEFTNODES  |
|*  6 |     INDEX RANGE SCAN               | RI_ndabg   |
|   7 |   SORT AGGREGATE                   |            |
|   8 |    NESTED LOOPS                    |            |
|   9 |     SORT UNIQUE                    |            |
|  10 |      COLLECTION ITERATOR PICKLER FETCH| RIGHTNODES |
|* 11 |     INDEX RANGE SCAN               | RI_nadbg   |
|  12 |   SORT AGGREGATE                   |            |
|* 13 |    INDEX RANGE SCAN                | RI_nadbg   |
-------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
6  - access(FNODE=LEFTNODES(value) AND DELTA>=832240.6 AND DELTA IS NOT NULL)
     filter(BETA*.2+.8*ALPHA<=835769.2 AND GAMMA*.2+.8*DELTA>=832240.6)
11 - access(FNODE=RIGHTNODES(value) AND ALPHA<=835769.2)
     filter(BETA*.2+.8*ALPHA<=835769.2 AND GAMMA*.2+.8*DELTA>=832240.6)
13 - access(FNODE>=832240.6 AND FNODE<=835769.2)
     filter(BETA*.2+.8*ALPHA<=835769.2 AND GAMMA*.2+.8*DELTA>=832240.6)
```

As the execution plan shows, for the two initial parts of the query $(2, 7)$, the RDBMS iterates (NESTED LOOPS) each node $(5, 10)$ obtained from transient tables *leftNodes* and *rightNodes*, respectively, scanning the RI indexes using the node value to access them and finally, retrieving those ones that satisfy the filter predicate $(6, 11)$, that is the selection criterium. The third part of the query is accomplished by means of an index range scan retrieving those tuples that satisfy the predicate 13.

5.2. Bitmap based indexing

This indexing strategy is based on the use of the following two bitmap indexes:

```
CREATE BITMAP INDEX BIT_ALPHA ON TAB(alpha); CREATE BITMAP INDEX BIT_DELTA ON TAB(delta);
```

The pre–selection step processing can be improved with the use of these indexes but, to perform the complete selection process, it is necessary to retrieve from the table the trapezoidal representation of the pre–selected values, given that it not practically feasible to create composed bitmaps indexes, contrary to what we have done with B-tree based indexes.

The rewritten query that forces the use of this index strategy is:

```
SELECT /*+ INDEX_COMBINE (TAB BIT_Alpha BIT_Delta)*/ count(*) FROM TAB
WHERE (alpha <=U_CT AND delta >=L_CT) AND (beta*T + alpha*(1-T)<=U_CT AND gamma*T+ delta*(1-T)>=L_CT)
```

The execution plan for an example query using this strategy is:

```
-------------------------------------------------
| Id  | Operation                   | Name      |
-------------------------------------------------
|   0 | SELECT STATEMENT            |           |
|   1 |  SORT AGGREGATE             |           |
|*  2 |   TABLE ACCESS BY INDEX ROWID | TAB       |
|   3 |    BITMAP CONVERSION TO ROWIDS|           |
```

```
|   4 |     BITMAP AND                |           |
|   5 |      BITMAP MERGE             |           |
|*  6 |       BITMAP INDEX RANGE SCAN | BIT_DELTA |
|   7 |      BITMAP MERGE             |           |
|*  8 |       BITMAP INDEX RANGE SCAN | BIT_ALPHA |
-------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
2 - filter("BETA"*0+1*"ALPHA"<=56832 AND "GAMMA"*0+1*"DELTA">=51832)
6 - access("DELTA">=51832); filter("DELTA">=51832); 8 - access("ALPHA"<=56832); filter("ALPHA"<=56832)
```

As the execution plan shows, the index BIT_ALPHA is scanned searching the bitmaps for all values of alpha which are less or equal than 56832 (8). Then, these bitmaps are merged using a bitwise OR (7). Similar processing is performed using the BIT_DELTA index (6,5). The two previous steps return two bitmaps on which the intersection is calculated by means on a bitwise AND (4). Then, the obtained bitmap is transformed into a set of row identificators (3) which allow to retrieve from the table the pre–selected tuples. On these tuples the "filter" predicate is applied to retrieve those ones that satisfy the selection criteria (2). This indexing strategy will be tagged as **BIT**.

5.3. R-Tree based indexing

To evaluate a strategy based on multidimensional indexes (that can be feasible to implement on traditional RDBMS), we have used Oracle Spatial©. This feature of Oracle© is based on the use of R-Trees to enhance spatial queries. To implement this multidimensional indexing strategy, we perform the following steps: first, we define the interval that represents the support of a fuzzy datum ([alpha,delta]) as a point in a bi-dimensional space, i.e., as an instance of the sdo_geometry type in this way:

```
sdo_geometry(2001, null, SDO_POINT_TYPE(alpha,delta,NULL), NULL, NULL)
```

where, value 2001 indicates that this instance is a point into a bi-dimensional space, the NULL value in the second parameter establishes the use of the cartesian coordinate system, the third parameter provides, by means of the SDO_POINT_TYPE constructor, an instance of the point with coordinates: x=alpha, y=delta and NULL for the z coordinate. The last parameters are not used in our case.

Then, we create the table that stores the fuzzy data represented in this way:

```
CREATE TABLE RTAB (support SDO_GEOMETRY, beta NUMBER, gamma NUMBER);
```

Next, we create a spatial index on the support attribute by means of the sentence:

```
CREATE INDEX RT_supp ON RTAB (support) INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

The insertion of fuzzy data ([alpha,beta,gamma,delta]) in this table is performed by means of the statement:

```
INSERT INTO RTAB VALUES(sdo_geometry(2001,null,SDO_POINT_TYPE(alpha,delta,NULL),NULL,NULL),beta,gamma);
```

Finally, it is necessary to rewrite each possibilistic query for the use of this indexing strategy in this way:

```
SELECT count(*) FROM RTAB WHERE SDO_FILTER(support,
 SDO_GEOMETRY(2003, NULL, NULL,SDO_ELEM_INFO_ARRAY(1,1003,3),SDO_ORDINATE_ARRAY(0,L_CT,U_CT,100000)))='TRUE'
AND (beta*.3 + .7*I.support.SDO_POINT.X <=U_CT AND gamma*.3 + .7*I.support.SDO_POINT.Y >=L_CT);
```

where, the SDO_Filter function delimits the pre–selection search using the defined spatial index. To do this, by means of the SDO_GEOMETRY type, a rectangle search defined by the left lower coordinates (0,L_CT) and the right upper coordinates (U_CT,100000) is defined; Note that, in our example, the domain range is [0,100000]. The last part of the query statement comprises the selection condition taking the necessary attributes values from the table.

The execution plan generated for a query example which uses this index strategy is:

```
-----------------------------------------------
| Id  | Operation                    | Name     |
-----------------------------------------------
|   0 | SELECT STATEMENT             |          |
|   1 |  SORT AGGREGATE              |          |
|*  2 |   TABLE ACCESS BY INDEX ROWID| RTAB     |
|*  3 |    DOMAIN INDEX              | RT_SUPP  |
-----------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
2 - filter("BETA"*.3+.7*"I"."SYS_NC00004$"<=4672.5 AND "GAMMA"*.3+.7*"I"."SYS_NC00005$">=1157.5)
3 - access("SDO_FILTER"("SUPPORT","SDO_GEOMETRY"(2003,NULL,NULL,"SDO_ELEM_INFO_ARRAY"(1,1003,3),
"SDO_ORDINATE_ARRAY"(0,1157.5,4672.5,100000)))='TRUE')
```

As the execution plan manifests, the pre–selection step is performed by means of the use of the multidimensional index through the "access" predicate (3). The rows identifiers retrieved in this step allow to access the values in the table to perform the selection step (2). This indexing strategy is labeled as **RT**.

Name	**Support Position Distribution**	**Support Length Distribution**
$D_1(n,d)$	uniform	uniform in $[0,2d]$
$D_2(n,d)$	*in* $[0,10^{20}-1]$	exponential in $[0,\infty]$, mean=d
$D_3(n,d)$	poisson process	uniform in $[0,2d]$
$D_4(n,d)$	*in* $[0,10^{20}-1]$	exponential in $[0,\infty]$, mean=d

Table 3: Types of databases generated for **DBset2**

6. Performance Evaluation

To determine the best indexing strategy we will evaluate its real performance by performing tests on a traditional RDBMS. The experimental setups have been developed on a Win64 version of the Oracle© 11.2 RDBMS running on a server having a Core i7 CPU with 4 cores running at 3.4GHz, 32 GB of RAM and a 512GB SSD as secondary storage.

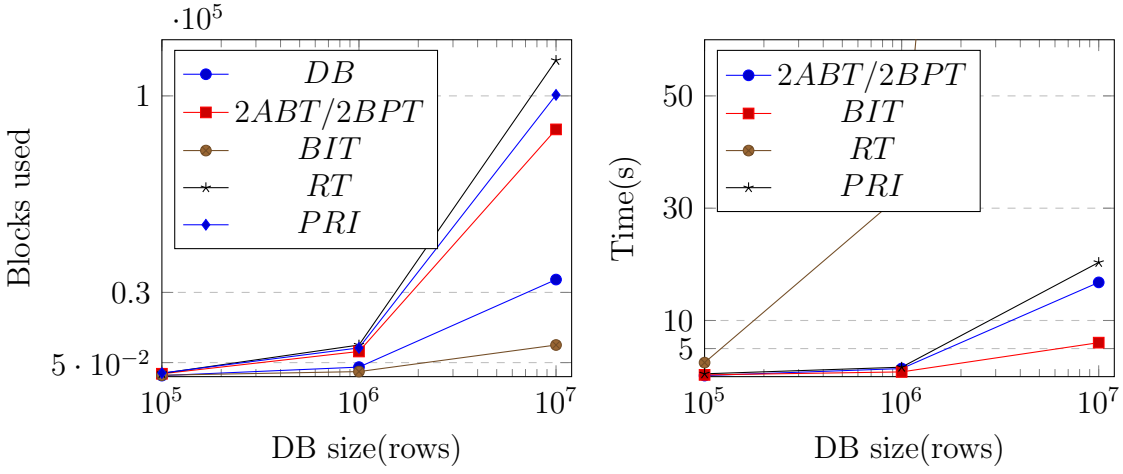
We have carried out several tests on two sets of databases, the first one for testing the performance and scalability of each indexing strategy and, the second one, to reproduce the same conditions that the used in the experiments evaluated in the work Kriegel et al. (2000), to compare our proposals in the same conditions. In total, we have generated 63 tables containing from 1000 to 10^7 tuples, with 249 index structures built on them and, finally, we have generated, executed and evaluated 2272262 queries on these tables. The database block size used is *8 KB* and, to get the same execution conditions, the database

block buffer cache was cleaned before each query execution (this forces all requested blocks to be recovered from the secondary storage).

The first set of databases (**DBset1**) was generated on the domain [0,100000] and comprises the following elements:

- 3 tables with 100000 tuples, containing trapezoids uniformly distributed with a fixed support size of 50, 500 and 5000, respectively. The kernel of each trapezoid was generated using also a uniform distribution.
- 3 tables with 1.000.000 tuples and 3 tables with 10^7 tuples which have been generated following the same distribution parameters that the previous ones.
- To evaluate the proposal based on the use of the RI-Tree strategy, 9 versions of the previous tables have been generated which, additionally, include the attribute fnode (which stores the forknode value for the support of the trapezoid)
- To evaluate the proposal based on the use of the R-Tree strategy, 9 versions of the above tables have been generated using a spatial representation of the support and the kernel of the trapezoids contained in these tables.
- The necessary indexes to implement each evaluated indexing strategy.

The second set of databases (**DBset2**) was generated on the domain $[0, 10^{20} - 1]$ and consists of 18 tables with different cardinalities (n) which contain trapezoids whose position and size (d) have been generated combining the uniform distribution and the Poisson process as summarized in Table 3.



(a) Size of the Indexes and Tables (Blocks)

(b) Time spent for index creation

Figure 5: Storage and time to indexes generation respect to DB size.(**DBset1**).

We have generated another set of 18 tables with the same data, respectively, which append the forknode attribute to allow testing of **PRI** indexing strategy. Also, the necessary indexing structures have been built.

6.1. Storage use and performance for index construction

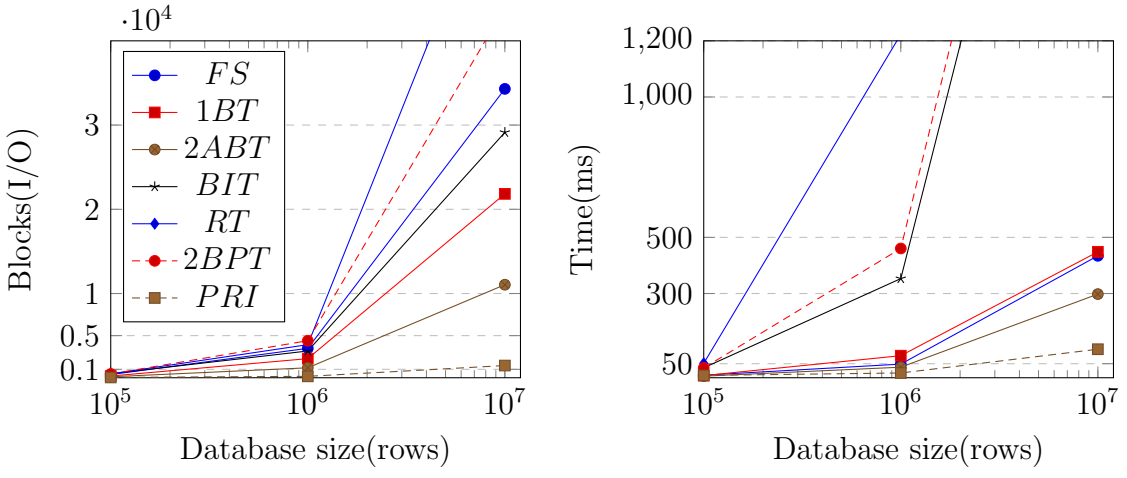
Fig. 5 a) shows the number of blocks used for each indexing strategy evaluated with respect to the amount of rows indexed. Also includes the size (blocks used) for each table size (**DB** label in Fig. 5 (a)). Note that **2BPT** use the same two indexes as the **2ABT** strategy that is shown in the figure. It must be also noted that for all the indexing strategies that use two index structures, the reported figures refer to the total size of them. The **1ABT** strategy only uses a index, because of this, the size and time spent is half of the ones used by the **2ABT/2BPT** strategies.

The size used for the **BIT** indexing strategy is the smallest because the bitmap index size depends on the number of different key values which, in this case, is 100.000 at most, not the number of indexed rows, like the other indexing techniques evaluated. Moreover, this technique, along with the **RT** technique, does not include in each index entry the rest of values of the trapezoid. The size used by the other indexing techniques grows linearly (approximately).

With respect to the time spent for the building of the indexes, Fig. 5 b) shows again that **BIT** is faster, **2ABT**, **2BPT** and **PRI** techniques grow linearly (approximately) with respect to the size of the DB and, the time spent for the **RT** technique increases exponentially with respect to the size of the DB.

6.2. Query performance tests

To take as reference the query performance when no indexing strategies are used, we have also performed the query tests directly on the base table performing a full scan (notated as **FS** in the figures). The structure of this base table is only composed by the four attributes that store the trapezoidal representation of the fuzzy data. It must be noted that, this simple structure, with no more attributes, performs better on full scan queries than real tables with more attributes, because it needs to retrieve fewer database blocks. Therefore, this setup represents the best case when comparing with the other indexing techniques.

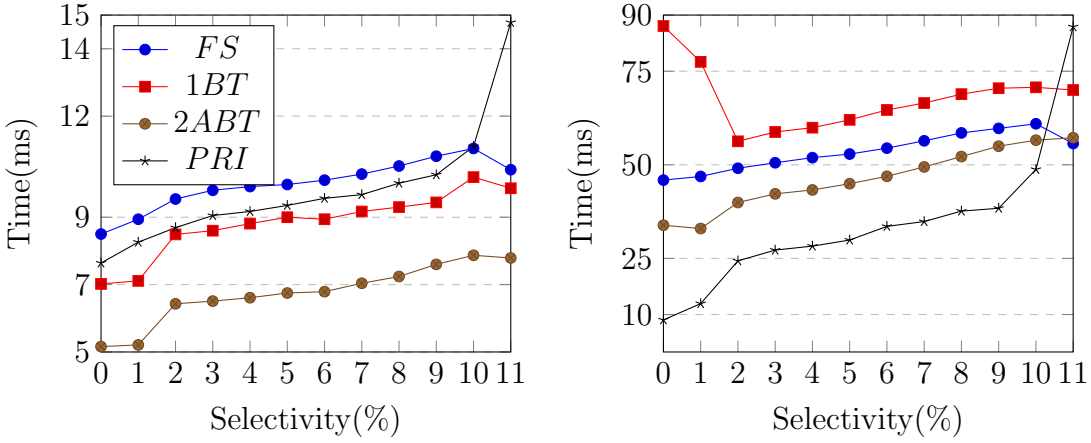


(a) Average blocks retrieved

(b) Average execution time

Figure 6: Average performance respect to database size (**DBset1**).

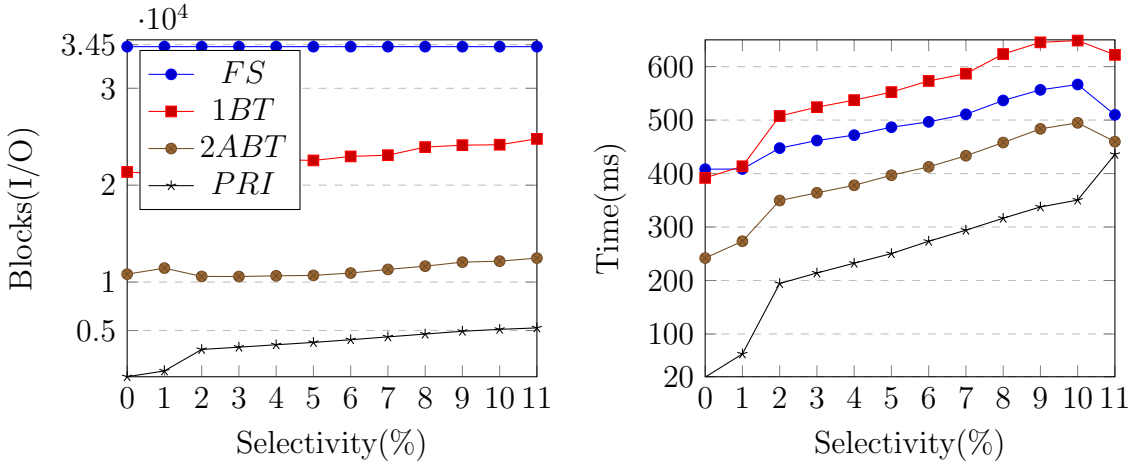
Figs. 6 to 9 show the results of tests performed on the database set **DBset1**. Fig. 6 shows the average performance respect to the size of the queried table. The **PRI** technique shows the better average performance with respect to the amount of blocks retrieved and with respect to the time spent in the execution of the queries, for all table sizes evaluated. The **2ABT** technique shows the second best performance. The average time spent for query execution for **BIT**, **RT** and **2BPT** techniques is very large, even compared with performing a full scan (**FS**). The average blocks retrieved also show the same trend. In the rest of our tests we get the same poor performance for this techniques; because of this, we will focus on the performance analysis of the rest of the techniques. The selectivity of the queries determines the strategy used by de QO in order to



(a) Database size 10e5 rows (b) Database size 10e6 rows

Figure 7: Average execution time spent with respect the query selectivity (**DBset1**).

consider, or not, the use of an index.

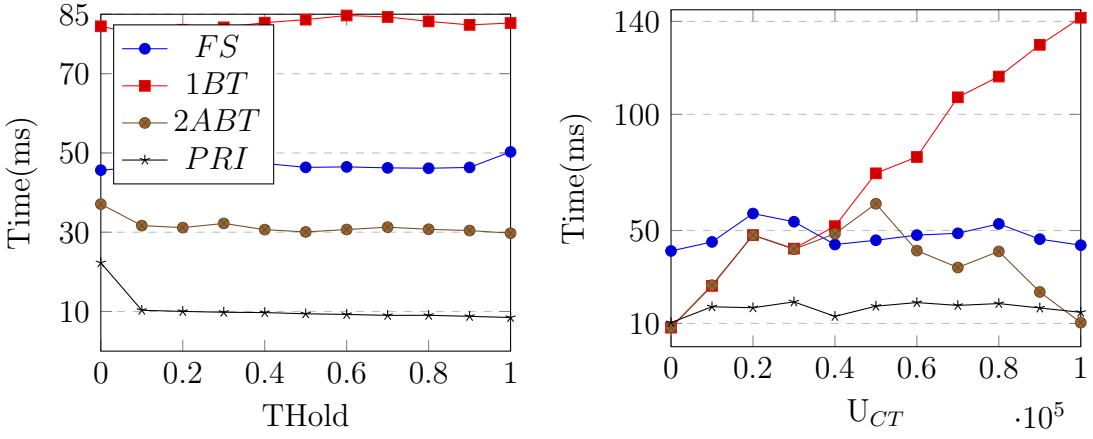


(a) Average blocks retrieved (b) Average execution time

Figure 8: Average performance respect to selectivity (Database size 10e7 rows) (**DBset1**).)

In general, for selectivities greater than the 10%, a full scan performs better than an

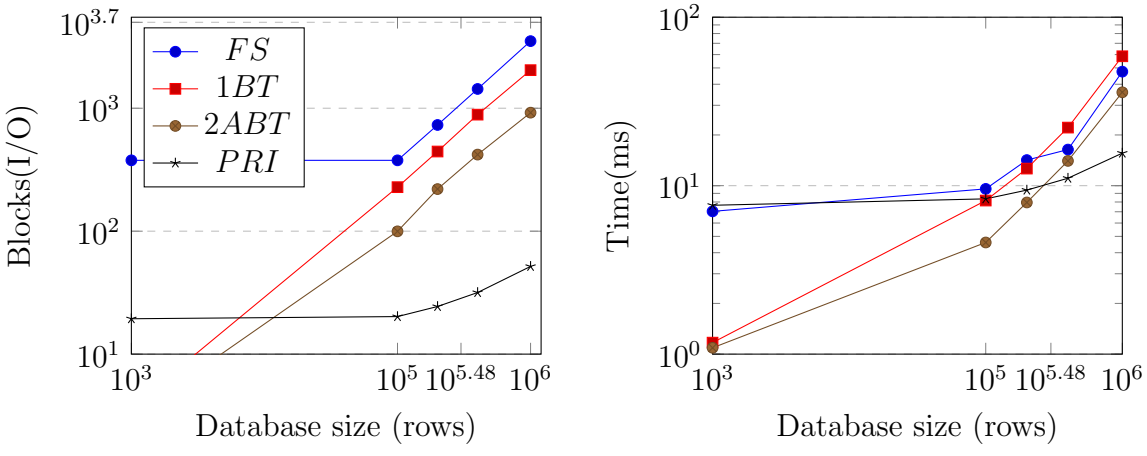
indexing based retrieval. Figs. 7 and 8 show the average performance with respect to the selectivity of the queries evaluated in databases of different size. As we can see, the **2ABT** indexing strategy performs better on databases with size near to 100000 tuples. The performance of the **PRI** technique for databases of this size is worse, even when compared to **1BT** technique. However, for larger database sizes the **PRI** technique outperforms the rest of techniques; this is due to the smaller amount of blocks required to execute the queries in databases of these sizes. This performance dependency with respect to the database size, shown by the **PRI** and **2ABT** techniques, remains in the rest of performed tests.



(a) 10e6 rows, selectivity 1 %

(b) 10e6 rows, selectivity up to 1%

Figure 9: Average execution time in function of threshold and U_{CT} (**DBset1**).



(a) Average blocks used

(b) Average execution time

Figure 10: Performance with respect to database size: $D_4(*, 2k)$, Selectivity 0.6%. (**DBset2**).

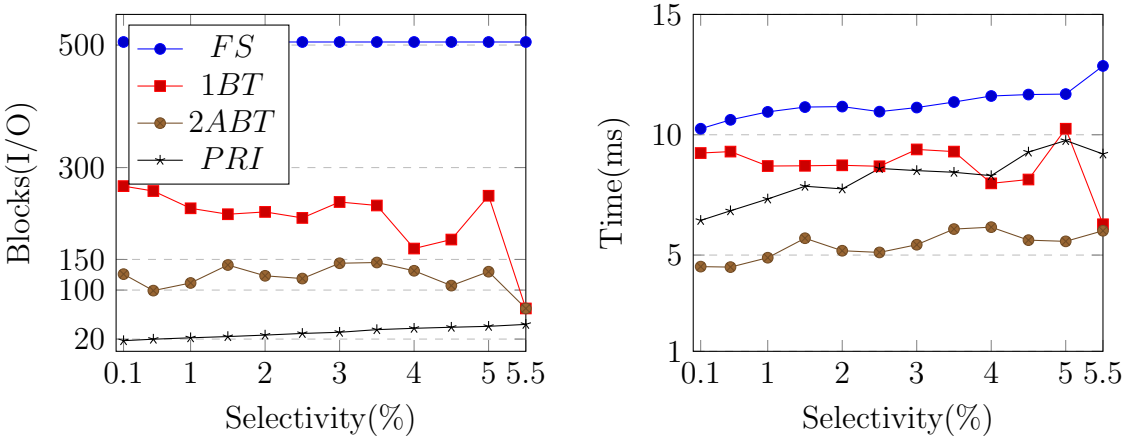
To conclude the query performance analysis on the database set **DBset1**, we analyze how the threshold set in the query affects the performance of the queries (Fig. 9 (a)) and how the query performance varies depending on the U_{CT} value (Fig. 9 (b)). As we can

see, the performance barely varies with respect to the threshold of the query, because the selection step is performed on the index in the case of the considered techniques. For the database size shown (10e6 tuples) the **PRI** technique is the best, followed by the **2ABT** technique. For a database size of 100000 tuples, the results are swapped between **PRI** and **2ABT** techniques.

As Fig 9 b) shows, the U_{CT} value affects the performance of the query for all indexing strategies except **FS** and **PRI**. The **2ABT** strategy performs worse for U_{CT} values around the middle of the domain, because it needs a larger scan on one of the defined indexes. Note how the efficiency of the **1BT** strategy decreases as U_{CT} grows; this is because the access condition ($alpha <= U_{CT}$) on the defined index on the alpha value retrieves more entries for large U_{CT} values.

Next, we analyze the results obtained for the tests performed on the database set **DBset2**. This database set and the tests we have executed on it have been built following the experimental set up in the work Kriegel et al. (2000) to compare with the indexing strategies analyzed there. For each test, we have generated and run an average of 100 queries.

The blocks used and the time spent by the indexes generation for each indexing strategy follows the same trend observed for the database set **DBset1**.



(a) Average blocks used

(b) Average execution time

Figure 11: Performance with respect to selectivity, $D_1(100k, 2k)$ (**DBset2**).

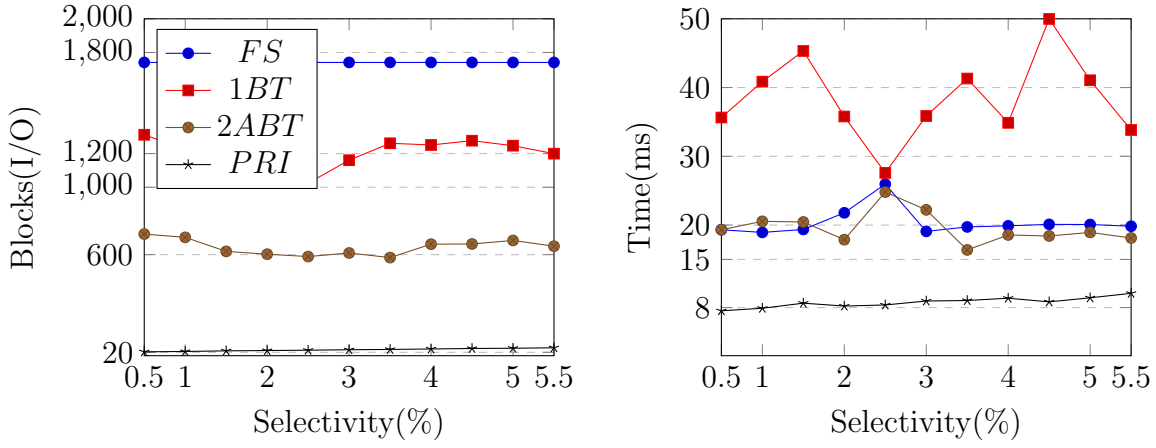
Fig. 10 shows the average query performance depending of the database size. For databases up to *200,000* tuples, the **2ABT** strategy shows the best overall performance. For larger databases, **PRI** outperforms the other techniques.

The selectivity of the query in the avobe test was set to 0.6%. Fig. 11 shows the results obtained on the database $D_1(100k, 2k)$ from different selectivity level of queries. For this database size, like in previous tests, **2ABT** strategy outperforms the **PRI** technique, although the **PRI** technique retrieves fewer blocks from the indexes. This is due to low density of selected trapezoids in each retrieved node and this fact increases the processing time.

When the test is carried out on a database with 400.000 tuples, as Fig. 12 shows, the results demonstrate a better performance for the **PRI** technique versus the **2ABT**

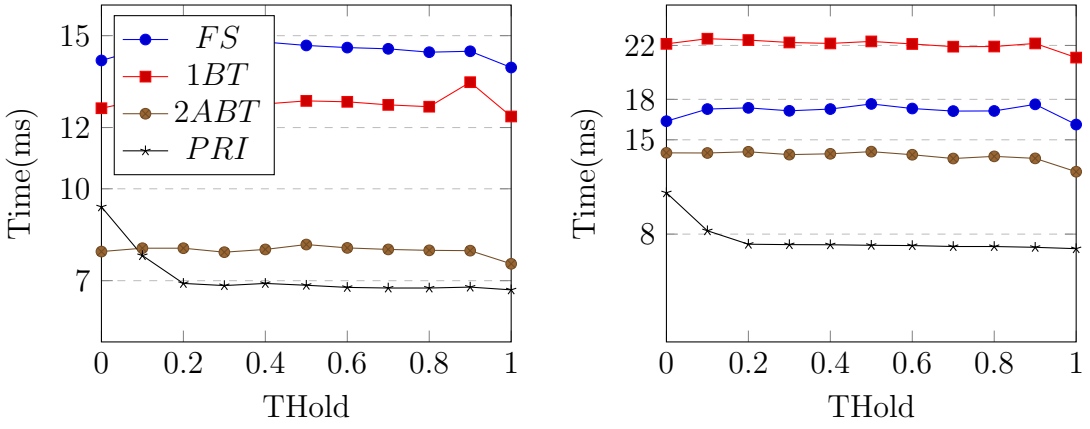
technique.

Fig. 13 shows that the threshold of the query does not affect the performance, because in the retrieval process prevails the pre–selection step performed in the index, and this depends marginally on the set threshold. For the database with 200.000 tuples, the **PRI** outperforms the **2ABT** technique except for queries with threshold below 0.1; in such case, most of the pre–selected trapezoids also satisfy the selection condition and it is manifested the best retrieval performance of the B-trees. However, for a database with 400.000 tuples, the performance of the **PRI** technique is better than the others.



(a) Average blocks used (b) Average execution time

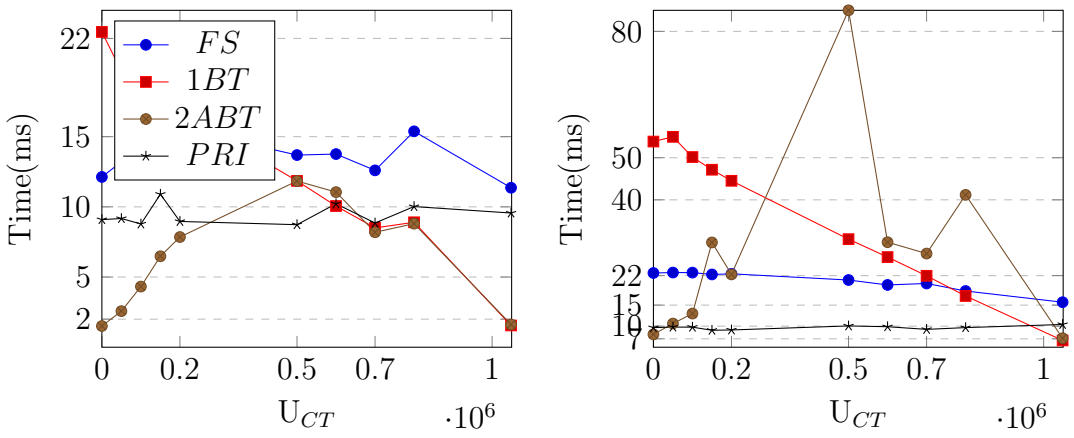
Figure 12: Performance with respect to selectivity, $D_1(400k, 2k)$ (**DBset2**).



(a) $D_4(200k, 2k)$ (b) $D_4(400k, 2k)$

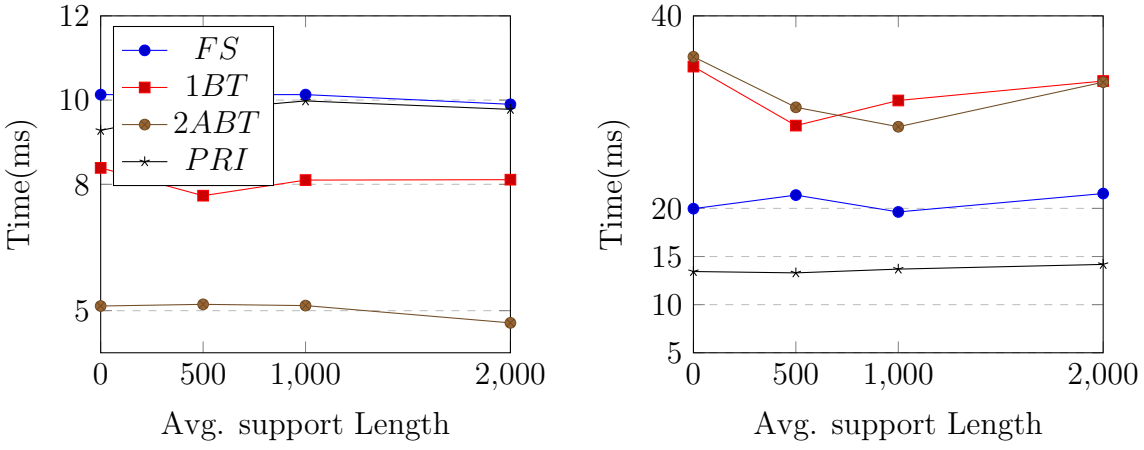
Figure 13: Average execution time with respect to the threshold (Selectivity 0.6%). (**DBset2**).

Fig. 14 shows how the U_{CT} value affects the performance of the query. Again, **2ABT** performs better on a database with (200.000) tuples (Fig. 14 a)), except when U_{CT} takes a value around the middle of the domain. For a database with 400.000 tuples, the **PRI** technique is more stable and performs better, except for those U_{CT} values around the



(a) $D_2(200k, 2k)$ (b) $D_2(400k, 2k)$

Figure 14: Average execution time with respect to U_{CT} position (Selectivity 0.2%). **DBset2**.



(a) D4(100k,*) (b) (D4(400k,*)

Figure 15: Average execution time with respect to the support length (Selectivity 1%). (**DBset2**).

bounds of the domain because, in this case, **2ABT** retrieves very few items from the selected index.

Finally, Fig 15 shows how the query performance depends on the average length of the indexed supports. Again, the size of the database evaluated determines the winner indexing strategy: **2ABT** for a database with 100.000 tuples, and the **PRI** technique for a database with 400.000 tuples.

6.3. Discussion

We have carried out an exhaustive set of tests covering a wide variety of databases and kind of queries for evaluating a lot of indexing strategies. All databases, indexing strategies and tests have been implemented, executed and evaluated in a real RDBMS, Oracle© in our case, using only the mechanisms and structures it provides.

From the analysis of the results obtained from the tests we can highlight the following:

- As occurs when indexing crisp data, the use of indexes only is efficient when applied on large tables and when the queries are selective enough. The designer must decide the use of an index for an estimated size of table and, on the other hand, the QO must decide to use or not an index depending of the selectivity of a given query.
- It is difficult to design an indexing strategy that enhances the performance of queries based on the search for overlapping intervals, which is basically our case.
- In fact, most indexing strategies evaluated perform worse than a full scan on the database. It is necessary to point that this full scan carried out on a table that contains only a column with trapezoidal data; a table with more attributes, probably decreases the efficiency of the full scan.
- The indexing strategies that do not include all trapezoidal values in the index entries do not perform efficiently because these ones require additional retrieval of database blocks to complete the selection step. This is the case of **BIT** and **RT** techniques.
- The indexing techniques that require the conjunctive composition of two sets of results, like **2BPT**, are computationally expensive because these sets require some kind of previous sorting.
- The indexing strategy that better performs on data sets whose size exceeds *200000* rows is **PRI**. Furthermore, this strategy is the most stable with respect to variations on the parameters of the query (the threshold, size and position of the queried trapezoid), with respect to selectivity of the query and the distribution of the trapezoids in the data set.
- For data sets whose size is less than *200000*, the **2ABT** indexing strategy performs better in general. However, its performance slightly decreases when the queried trapezoid is located about the middle of the domain.

7. Concluding Remarks and Future Works

The goal of our work is to find the most efficient indexing techniques for possibilistic queries on fuzzy databases, using the access structures and the extension capabilities available on a classical RDBMS only. To carry out this task, we have proposed, adapted, implemented and tested thirty indexing strategies variants using the most important indexing structures available in classical RDBMS: B-trees, bitmaps and R-trees. Along them, we have considered the most relevant proposals in the literature, modified some of these approaches, and proposed some novel ones.

The results of this experimental work can be summarized as follows

- It is feasible and efficient to implement indexing techniques to enhance possibilistic query retrieval using indexes available in traditional RDBMS. One of the best evaluated indexing techniques, our **2ABT** technique, consists of two B-trees and an optional query optimiser algorithm, which can be implemented on any classical RDBMS. The other best indexing technique, which performs better on large databases, the **PRI** technique, needs to implement additional structures to operate, but it is relatively easy to implement also.

- One of the most important factors for performance retrieval in RDBMS is the amount of database blocks the query resolution involves, not the amount of items accessed. Because of this, some indexing techniques suitable for searching in memory, do not perform well on the database scope.

Future works will focus on integrating the best evaluated indexing strategies, **2ABT** and **PRI**, in our FORDBMS prototype (Barranco et al. (2008c)) built on Oracle© RDBMS. Also, we will perform a cost analysis for its integration into the QO for the selection of the better execution plan for each query. Finally, we will perform a similar study for necessity based queries, to find the best indexing strategy to be used on classical RDBMS also.

Acknowledgement

This work has been partially supported by the Spanish Ministry of Science under grant **TIN2014-58227-P**.

References

Barranco, C. D., Campaña, J. R., & Medina, J. M. (2008a). A B^+-tree based indexing technique for fuzzy numerical data. *Fuzzy Sets and Systems*, *159*, 1431–1449. URL: http://www.sciencedirect.com/science/article/B6V05-4RP0MFH-3/2/0352e55fbd32e91ea27069c6b8270e29.

Barranco, C. D., Campaña, J. R., & Medina, J. M. (2009). Indexing fuzzy numerical data with a B+tree for fast retrieval using necessity-measured flexible conditions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *17 – Supplementary issue 1*, 1–23. URL: http://www.worldscinet.com/ijufks/17/17supp01/S0218488509006005.html. doi:10.1142/S0218488509006005.

Barranco, C. D., Campaña, J. R., & Medina, J. M. (2008b). A b+-tree based indexing technique for fuzzy numerical data. *Fuzzy Sets and Systems*, *159*, 1431–1449.

Barranco, C. D., Campaña, J. R., & Medina, J. M. (2008c). Towards a fuzzy object-relational database model. In J. Galindo (Ed.), *Handbook of Research on Fuzzy Information Processing in Databases* (pp. 435–461). IGI Global.

Barranco, C. D., & Helmer, S. (2012). An impact ordering approach for indexing fuzzy sets. *Fuzzy Sets and Systems*, *196*, 33 – 46. URL: http://www.sciencedirect.com/science/article/pii/S0165011411000558. doi:http://dx.doi.org/10.1016/j.fss.2011.01.014. On Advances in Soft Computing Applied to Databases and Information Systems.

Bosc, P., & Galibourg, M. (1989). Indexing principles for a fuzzy data base. *Information Systems*, *14*, 493 – 499. URL: http://www.sciencedirect.com/science/article/pii/0306437989900173. doi:http://dx.doi.org/10.1016/0306-4379(89)90017-3. Fuzzy Databases.

Bosc, P., & Pivert, O. (1995). Sqlf: a relational database language for fuzzy querying. *Fuzzy Systems, IEEE Transactions on*, *3*, 1–17. doi:10.1109/91.366566.

Caluwe, R. D. (1997). *Fuzzy and Uncertain Object-Oriented Databases: Concepts and Models* volume 13 of *Advances in Fuzzy Systems-Applications and Theory*. World Scientific.

Edelsbrunner, H. (1983). A new approach to rectangle intersections. *International Journal of Computer Mathematics*, *13*, 209–229.

Fukami, S., Umano, M., Muzimoto, M., & Tanaka, H. (1979). Fuzzy database retrieval and manipulation language. *IEICE Technical Reports, AL-78-85 (Automata and Language)*, *78*, 65–72.

Galindo, J., Medina, J., Pons, O., & Cubero, J. (1998). A server for fuzzy sql queries. In T. Andreasen, H. Christiansen, & H. Larsen (Eds.), *Flexible Query Answering Systems* (pp. 164–174). Springer Berlin Heidelberg volume 1495 of *Lecture Notes in Computer Science*. URL: http://dx.doi.org/10.1007/BFb0055999. doi:10.1007/BFb0055999.

Kacprzyk, J., & Zadrożny, S. (1995). Fquery for access: Fuzzy querying for a windows-based dbms. In P. Bosc, & J. Kacprzyk (Eds.), *Fuzziness in Database Management Systems* (pp. 415–433). Physica-Verlag HD volume 5 of *Studies in Fuzziness*. URL: dx.doi.org/10.1007/978-3-7908-1897-0_18. doi:10.1007/978-3-7908-1897-0_18.

Kriegel, H.-P., Pötke, M., & Seidl, T. (2000). Managing intervals efficiently in object-relational databases. In *Proc. 26th Conf. on Very Large Data Bases (VLDB 2000). Cairo (Egypt)* (pp. 407–418).

Liu, C., Ouksel, A., Sistla, P., Wu, J., Yu, C., & Rishe, N. (1996). Performance evaluation of g-tree and its application in fuzzy databases. In *CIKM '96: Proceedings of the fifth international conference on Information and knowledge management* (pp. 235–242). New York, NY, USA: ACM Press. doi:http://doi.acm.org/10.1145/238355.238503.

Medina, J. M., Pons, O., & Vila, M. A. (1994). Gefred. a generalized model of fuzzy relational databases. *Information Sciences*, *76*, 87–109.

Oracle (2012). Performance tuning guide. In *Oracle Database v. 11.2. Documentation* (pp. 1–560). Available in: http://docs.oracle.com/cd/E11882_01/server.112/e41573/toc.htm.

Prade, H., & Testemale, C. (1984). Generalizing database relational algebra for the treatment of incomplete or uncertain information and vague queries. *Information Sciences*, *34*, 115–143.

Testemale, C. (1986). Fuzzy relational databases—a key to expert systems. *Journal of the American Society for Information Science*, *37*, 272–273. URL: http://dx.doi.org/10.1002/(SICI)1097-4571(198607)37:4<272::AID-ASI15>3.0.CO;2-U. doi:10.1002/(SICI)1097-4571(198607)37:4<272::AID-ASI15>3.0.CO;2-U.

Umano, M. (1982). Freedom-o: A fuzzy database system. In Gupta-Sanchez (Ed.), *Fuzzy Information and Decision Processes* (pp. 339–347). North-Holand.

Yazici, A., Ince, C., & Koyuncu, M. (2008). Food index: A multidimensional index structure for similarity-based fuzzy object oriented database models. *Fuzzy Systems, IEEE Transactions on*, *16*, 942–957. doi:10.1109/TFUZZ.2008.917304.