Received: 24 June 2021 | Accepted: 13 December 2021

DOI: 10.1002/int.22807

RESEARCH ARTICLE

WILEY

A fuzzy database engine for mongoDB

Juan Miguel Medina | **Ignacio J. Blanco** | **Olga Pons**

Department of Computer Science and Artificial Intelligence, University of Granada, Granada, Spain

Correspondence
Juan Miguel Medina, Department of Computer Science and Artificial Intelligence, University of Granada, C/Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain.
Email: medina@decsai.ugr.es

Funding information
Universidad de Granada

Abstract

Big Data are a paradigm through which valuable information is achieved through the analysis of a large amount of data. The sources of these data can be varied, from data streams that will be processed in real time, to the exploitation of transactional data stored in databases. For this last use, due to their scalability, the NoSQL databases, like mongoDB, a DBMS oriented to documents, have been consolidated as a powerful tool for the storage and processing of large volumes of data. On the other hand, information sources for Big Data algorithms can contain imprecise information, and the way to obtain, aggregate and present results can have an imprecise nature as well. For this reason, it is useful to provide fuzzy extensions to these DBMSs. In the case of MongoDB, there are few proposals and not very complete. This paper describes *fzMongoDB*, a fuzzy database engine that provides the mongoDB database with the capacity to store documents with imprecise information and to retrieve them in a flexible way. It is implemented and integrated on the mongoDB server using the resources it provides. The model and implementation of *fzMongoDB* also includes an indexing mechanism that accelerates the retrieval process on fuzzy queries. Also, the performance of these indexing mechanisms is evaluated.

KEYWORDS
fuzzy databases, fuzzy NoSQL databases, mongoDB

1 | INTRODUCTION

Currently, the Big Data paradigm is one of the most active research areas and there are many advances in each of its components. The goal of this paradigm is the extraction of valuable information (value) from a large amount of data (volume) with heterogeneous structure (variety) and investing a reasonable amount of time (velocity). But there are other factors involved in this study area, such as the *validity* of the data to be processed and the results obtained, and the way the starting data are processed and how the results are shown. These factors can benefit from the use of techniques in the Soft Computing area. In this sense, several works such as References [1–3] propose the use of fuzzy models to handle vagueness in Big Data processing, especially in relation to the handling of the variety and veracity inherent to the available data.

In the Big Data paradigm, database management systems (DBMSs) play an important role because they provide the storage mechanism for data streams and for the transactional data involved in ulterior processing. Specifically, the NoSQL DBMSs are of interest for this paradigm, due to its features like scalability and suitability for the big data algorithms processing. NoSQL is a general category of DBMSs that do not follows the basics of the relational model. This category comprises several kinds of DBMS, Reference [4] classifies five kinds of NoSQL DBMSs: key-value-based databases, column-oriented databases, document-oriented databases, graph-based databases and object-oriented databases. Our work focuses on document-oriented DBMS, specifically on mongoDB[5] DBMS.

To deal with imprecise data in the Big Data context, these database systems need the capability to represent and handle fuzzy data. In the literature there are many proposals to provide to databases with the capability of handling imprecise data. Most of them have been focused on the application of the fuzzy handling to the relational model. Two approaches have been proposed to address the problem of imprecise treatment in these databases. The first approach consists on extensions of the SQL query language to perform fuzzy queries on classical relational databases. Some works that follow this approach are References [6–8].

The second approach extends the relational model toward a fuzzy relational model. The proposals in this category provide mechanisms to represent several kinds of fuzzy data into the models, and define the operators to perform fuzzy queries on these fuzzy data and on classical data too. Some proposals in this sense are References [9–11].

Also we can found several proposals that define fuzzy databases models and implementation prototypes based on other classical models. There are several proposals for fuzzy object databases, some of these are reviewed in References [12,13]. We have proposed in Reference [14] a model to provide fuzzy capabilities to an object-relational database system and an implementation for this on the ORDBMS Oracle.

Reference [13] compiles many proposals of models and implementations to enhance database models with fuzzy handling capabilities. Also, Reference [15] provides a review of the most interesting proposals in this sense.

However, there are only a few proposals that extend the capacity of NoSQL databases to deal with fuzzy data. Reference [16] shows a way to transform a fuzzy relational model into an HBase model (this is a database oriented to columns). Reference [17] proposes a framework (Fuzzy4S) and a fuzzy extension of the Cypher language (Cypherf) to provide Neo4J, which is a graph-based NoSQL database, with the ability to deal with fuzzy information.

When we search proposals to handle fuzzy information in document-oriented databases, we find a few proposals, fundamentally focused on the mongoDB[5] DBMS. This document-oriented

DBMS, which is the most used NoSQL database,[18] provides a public access version called mongoDB community edition. By analysing its features, we find that it is a good candidate to integrate capabilities for the fuzzy data storage with the support for the processing of these data using Big Data algorithms improved with techniques from the Soft Computing discipline. Reference [19] proposes a basic fuzzy extension of the mongoDB query language that allows one to query collections in a fuzzy way by means of the use of linguistic labels. Nevertheless, it does not allow one to represent fuzzy information in the database. In Reference [20], a fuzzy extension of the mongoDB language named FMQL is used to operate with fuzzy data. The proposed system translates operations expressed in this language into F-XML expressions. The problem is the lack of explanation on how the mongoDB system performs the necessary transformations from FMQL to F-XML and how mongoDB executes this translated code. Astachova et al. [21] present a proposal based on the implementation of a parser that translates fuzzy queries, expressed by means of an extension of SQL, to an expression on Mongo query language to be executed on it. The mongoDB database stores collections that can contain fuzzy data and linguistic values using a geometry representation based on the use of GeoJSON type. The fuzzy predicates on these data are computed through spatial operations on the arguments.

A most elaborate proposal to provide fuzzy handling on documents (JSON documents) is described in Reference [22]. In this paper the framework J-CO is proposed to collect, integrate, process, store and query JSON documents, which could be geotagged. The framework uses mongoDB to store the JSON documents. To operate with these documents the framework provides the J-CO-QL language. This language allows one to perform some kind of flexible query on JSON documents by means of the definition of "fuzzy operators" that are linguistic labels with their respective membership functions. Note that this framework does not store fuzzy information into the documents, only allows flexible queries about these.

In this paper we present *fzMongoDB*, a fuzzy module for extending mongoDB, by providing the capability to represent and handle fuzzy data. All the extended capacities are supported thanks to a fuzzy extension of the mongoDB language with a syntax consistent with the classical one, which allows one to integrate fuzzy clauses with classical ones. The implementation of this extension has been carried out using the resources provided by the mongoDB system, translating the fuzzy statements into mongoDB expressions so they can be directly executed. This module also provides an indexing mechanism that speeds up the retrieval process for queries based on possibility and on necessity.

The paper is organised as follows. In Section 2 the main elements of the MongoDB database are summarised, including the fundamental expressions and statements. Section 3 describes the fuzzy database model on which the *fzMongoDB* module is based. Next, in Section 4, the proposed syntax for this fuzzy extension is described. Section 5 describes how the *fzMongoDB* module is implemented and illustrates, by means of an example, how to create, populate and query a collection containing fuzzy data. In Section 6, it is shown a study of the performance of the indexing strategy proposed. Finally, Section 7 contains the conclusions and the future lines of research.

2 | MongoDB FUNDAMENTALS

MongoDB is an open-source document-oriented database and currently, it is the most used NoSQL database.[18] In this section we will review its main characteristics, its data model, the basic *C*reate, *R*ead, *U*pdate and *D*elete (CRUD) operations and the tools and resources it provides for implementing a fuzzy extension of the mongoDB system. Most of this information is extracted from the MongoBD documentation.[23]

The main characteristics of mongoDB are:

- High performance. It is written in C++, supports complex embedded documents, what reduces the number of *I*/*O*, and provides several types of indexes to increase the retrieval performance.
- A complete query language that supports, in addition to the basic CRUD operations, handling for geospatial and text search queries and aggregation queries that benefit from Big Data strategies.
- High Availability, through the mongoDB's replication facility, that provides automatic failover and data redundancy.
- Horizontal Scalability; by means of sharding, the data are distributed across a cluster of machines.
- *A*tomicity, *C*onsistency, *I*solation and *D*urability (ACID) support. From version 4.0, mongoDB can guarantee the ACID database properties.

2.1 | Data model

MongoDB provides a flexible data model, what allows us to store from a simple document of key-value pairs, up to a document with embedded subdocuments and/or complex arrays with various levels of depth. Data are stored in BSON format,[24] which is an extension of the native objects of JavaScript and provides some additional data types. Amongst these data types, several types of numerical data, strings, dates and timestamps, arrays, booleans, null, and ObjectID are included.

2.2 | CRUD operations

In mongoDB, all database operations are performed by means of invocation of database commands, through the statement `db.runCommand()`. MongoDB also provides shell, an interactive JavaScript interface composed of a set of methods for invoking commands of the database. Our proposal for a fuzzy extension includes a syntax based on these methods; because of this, in this section we will review the methods involved in the basic operations on a database: CRUD.

Data are stored in mongoDB through collections, which are comprised of a set of documents. It is not necessary to explicitly create a new collection since the execution of the method that inserts a new document into a collection, implicitly creates the new collection if it does not exist. For instance, let us suppose we want to create a collection of housings for real estate purposes. The next statement creates the housings collection and inserts a new document into it:

```
db.housings.insert(
{id: 301,
   type: "Flat",
   price: 129217,
   rooms: 2,
   area: 60,
   description: "CHURRIANA DE LA VEGA (Granada), 60 m2, 2 rooms, 1 bathroom, furnished
        kitchen, lift, exterior, garage, year 2005. Tlf. 958 ********"
})
```

It is possible to insert several documents by a single `insert` command, using an array of documents (a list of documents separated by commas enclosed in square brackets).

For retrieval purposes (*R*ead), mongoDB provides the database method `find`, whose basic syntax is as follows:

```
db.<collection>.find(<query>,<projection>)
```

where `<collection>` determines the collection to be queried, the `<query>` parameter is a document and allows one to express several types of queries: key-value, by range, compound queries, on fields that contain an array, on the array of documents, and so forth. This parameter contains basic comparison operations which are expressed in the way:

```
{<field>: {<rel_op>: <value>}}
```

where the symbols $eq, $ne, $lt, $gt, $lte, $gte for `<rel_op>` are used to express the relational comparators $=, \neq, <, >, \leq, \geq$, respectively. Note that `{<field>: {$eq: <value>}}` is equivalent to `{<field>: <value>}`.

To combine atomic comparison operations through the logical operators AND, OR and NOT, the following syntax is used, respectively:

```
{$and: [{<expression1>},{<expression2>}, ..., {<expressionN>}]}
{$or: [{<expression1>}, {<expression2>}, ..., {<expressionN>}]}
{field: {$not: {<operator-expression>}}}
```

The `<projection>` parameter is used to determine which fields appear or not in the query results; its basic syntax is

```
{<field1>: <0_or_1>, ..., <fieldN>: <0_or_1>}
```

To indicate that a field appears in the result of the query, the value must be set to 1; to explicitly exclude a field, the value must be set to 0. In the following example we illustrate the use of the *find* method to express a query on the housings collection.

Example 1. "Show the flats, with its price, area, number of rooms and description, which have three rooms and the price is less than or equal to 120000 euro."

```
db.housings.find({type: "Flat", rooms: 3, price: {$lte: 120000}},
                 {id:1, price:1, area:1, rooms:1, description:1})
```

The `update` method provides the functionality to change the contents of a document. Its basic syntax is as follows:

```
db.<collection>.update(<query>,<update>,
   {
     upsert: <boolean>,
     multi: <boolean>,
     //other options
   })
```

where `<collection>` indicates the collection to be updated, the `<query>` parameter is a document to determine which documents will be updated; this parameter uses the same syntax of the `<query>` parameter in the *find* method; the `<update>` parameter is a document that establishes the modifications to be applied. If the optional parameter `upsert` is set to true, a document is inserted when no document matches the query; if the optional parameter `multi` is set to *true*, the method updates all documents that match the query; if it is set to *false*, only one is updated.

The `deleteOne` method is used to remove a document from a collection; its syntax is

```
db.<collection>.deleteOne(<query>,
   {
      //optional options.
   }
)
```

where `<collection>` indicates the collection from which the document will be deleted; the `<query>` parameter is a document to determine the document to be deleted; this parameter uses the same syntax of the `<query>` parameter in the *find* method.

To delete all documents that match the query, the `deleteMany` method is provided. This method uses the same syntax as the `deleteOne` method. To delete all documents from a collection an empty document (`{}`) must be passed to the `<query>` parameter.

2.3 | Aggregation operations

Aggregation is one of the main proposals of mongoDB to process and transform data from collections. MongoDB provides two ways to carry it out, by means of the database commands `mapReduce` and `aggregate`.

2.3.1 | MapReduce command

The `mapReduce` command implements the MapReduce programming model, whose main features are the capacity to parallelise and distribute the processing, so it is frequently used in the Big Data area. The basic syntax for this command is

```
  db.runCommand({
  mapReduce: <collection>, //The collection to process
  map: <function>,
  reduce: <function>,
  finalize: <function>,
  out: <output>,
  query: <document>,
  //Others optional fields
})
```

In this command, the `map` field accepts a *JavaScript* function that associates or “maps” a value with a key and returns the key and value pair. In the `reduce` field it is necessary to attach a *JavaScript* function that "reduces" to a single object all the values associated with a particular key. If used, in the `finalize` field we can provide a *JavaScript* function that processes the output of the `reduce` method to modify the output. In the `out` field we specify where to store of the result of the map-reduce operation. In the `query` field we can determine which documents input the map function, this parameter uses the same syntax as the used for the `filter` clause of the `find` command.

2.3.2 | Aggregate command

This command uses an *aggregation pipeline* that consists in taking a collection as input and applying a sequence of stage-based manipulations on it. This command is fundamental for our implementation purposes, because we use it to execute the mongoDB expressions obtained after translating the fuzzy sentences into native mongoDB expressions, directly executable by this command. The basic syntax of this command is

```
    db.runCommand({
    aggregate: "<collection>",
    pipeline: [ <stage>, <...> ],
    cursor: <document>,
    //Others optional fields
  })
```

where in the `aggregate` field it is set the name of the input collection for the pipeline. To handle the output of the aggregation command a cursor object is used; by means of the cursor field, the options for the creation of this cursor objects are set. For instance, to indicate a cursor with the default batch size, it is necessary to set `cursor: {}`. In the `pipeline` field it is provided an array of aggregation pipeline stages that process and transform sequentially the document stream as part of the aggregation pipeline. Currently, there are 23 types of stages that can be used in the aggregation pipeline; these ones are full documented in Reference [23], but for our implementation purposes, we are mainly interested in the *match* and *project* stages.

The *match* stage filters the document stream, supplying to the next stage only those documents that match the specified query. The syntax is `{$match: {<query>}}`, where to express the `<query>` document it is used the same syntax than the used for the `filter` field of the `find` command. In addition to the relational and logical operators described in Section 2.2, we will describe those ones useful for our implementation:

- *$add*: An arithmetic operator that receives a list of expressions, which must be evaluated as a number, and adds all of them.
- *$substract*: An arithmetic operator that receives a list of expressions, which must be evaluated as a number, and subtracts all of them.
- *$multiply*: An arithmetic operator that receives a list of expressions, which must be evaluated as a number, and multiplies all of them.
- *$divide*: An arithmetic binary operator that receives a list of two expressions and returns the division of the first of them between the second.
- *$cond*: A conditional ternary operator that receives a list of three expressions and evaluates the condition expressed in the first expression, if it is evaluated as *true* executes the second

expression and if it is evaluated as *false* the third expression is executed. This operator allows us to perform the classic *if–else* of any programming language.
- *$switch*: An operator that evaluates cases expressions. It has the following syntax:

```
$switch: {
   branches: [
      { case: <expression1>, then: <expression1b> },
      { case: <expression2>, then: <expression2b> },
      ...
   ],
   default: <expression_D>
}
```

If `<expression1>` evaluates to *true*, `$switch` executes `<expression1b>`, the same for `<expression2>`, if it evaluate to *true* executes `<expression2b>`, If none expressions evaluate to true, `$switch` executes `<expression_D>`.
- *$arrayElemAt*: An operator for expressions with arrays. It receives a list of two expressions, where the first parameter is the field that corresponds to the array and the second one is the index of the array you want to obtain. It returns the value that is in the requested position of the array.
- *$type*: Returns a string that identifies the *BSON* type of the argument.
- *$expr*: Allows the use of aggregation expressions in the query language. Its use in combination with the operator "cond" is a powerful tool to elaborate complex conditional operations into the match stage.

The *projection* stage is similar to the *projection* parameter of the *find* method, except that it can generate new fields in the output collection using complex expressions that compute the desired results from the data in the fields of the input collection. To add a new field or to reset the value of an existing field, the following specification is used:

```
<field>: <expression>
```

where `<field>` is the new field to be added or reset and, `<expression>` is a document that can use a large number of aggregation pipeline operators. These include the same operators that can be used in the match stage.

2.4 | Query optimisation by indexing

Indexing is a mechanism provided by the DBMS to increase the performance of the retrieval processes. MongoDB provides several types of indexes: single and compound indexes, multikey indexes on arrays, geospatials and text indexes, and so forth.

For our implementation purposes, the more suitable index type to enhance the retrieval performance is the compound index. This one consists of an index structure that stores references to multiple fields, improving the queries based on them. The basic syntax for creating such indexes is

```
db.collection.createIndex({<field1>: <type>, <field2>: <type2>, …})
```

where the `<typen>` parameters indicate whether the index is arranged ascending (1) or descending (−1) in the corresponding field. Note that the order of the fields in the declaration is relevant for the index structure created and for the way the index is used in the query process.

MongoDB provides the *explain* helper to show the execution plan selected to execute a sentence. To show the execution plan of a sentence in its optional parameters, the document `{explain: true}` must be added. Then, a document is returned describing what statement will be executed, which indexes will be used (if defined), how they are used, and all the information that allows us to evaluate how the sentence will be executed.

Another resource to evaluate and optimise queries is the *hint* parameter. This parameter can be added to a query statement to determine the use of a specified index by the execution plan. The format to use this option is `{hint: <index>}` where `<index>` can be the index name, the index specification or `{$natural: <1_or_1>}` to force the query to perform a collection scan (1, for forward) (−1 for reverse).

The *explain* helper and the *hint* option will help us to evaluate the performance of the indexing strategy used for optimising queries in our mongoDB fuzzy extension.

3 | THE FUZZY DATABASE MODEL

The fuzzy database model that underlies the proposed fuzzy extension of the mongoDB system is based on the fuzzy relational database model described in References [10,25]. The fuzzy data model, the fuzzy comparison operators, and the fuzzy relational operators are adapted from this fuzzy relational model to the document-based database model of mongoDB. In this section, we will describe such components of this model.

3.1 | Fuzzy data

The proposed fuzzy database engine provides fuzzy treatment to several kinds of data: fuzzy numbers, scalars, unknown and undefined. We use a trapezoidal representation for the fuzzy numbers, given by Equation (1):

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

We can also represent crisp values and crisp intervals $[a, b]$ (which can be queried by means of fuzzy predicates). Table 1 summarises the fuzzy data supported for our fuzzy database engine, and how they are represented using the supported mongoDB data types.

TABLE 1 Fuzzy data types representation on mongoDB

Type	Description	Representation	Example
Crisp	A single number	`numericalvalue`	`area: 60`
Scalar	A single value on a not ordered domain with a nearness relation defined on its values	`"#"‖string`	`type:"#Flat"`
Linguistic label	A label that identifies a fuzzy number	`"$"‖string`	`price:"$Cheap"`
Interval	An interval value	`[<num>,<num>]`	`rooms: [2,3]`
Triangular	A triangular pos. distribution	`[<num>,<num>, <num>]`	`area:[50,60,65]`
Trapezoidal	A trapezoidal pos. distribution	`[<num>,<num>,<num>,<num>]`	`price:[1e5,1.2e5,1.3e5,1.5e5]`
Undefined	A nonapplicable value	`"$undefined"`	`type:"#Flat",floors: "$undefined"`
Unknown	An unknown value	`"$unknown"`	`area:"$unknown"`
Null	An undefined or unknown value	`null`	`type: "$unknown",floors: null`

To understand the datatypes in Table 1, we should clarify the following:

- The *Scalar* datatype refers to data defined on a scalar domain without an order relation defined on it. According to the GEFRED model,[10] to perform flexible queries on this datatype it is necessary to define a *nearness relation* on the domain values. To distinguish *Scalar* datatype from *string* datatype in mongoDB, we set the label with the # character.
- *Interval*, *Triangular* and *Trapezoidal* datatypes are particular representations of trapezoidal distributions; in mongoDB we use the array datatype to represent them.
- The *Linguistic label* datatype allows us to create labels that identify fuzzy numbers defined by trapezoidal distributions. In mongoDB, we distinguish this datatype from the *string* datatype prefixing the label with the $ character.
- The *Unknown* type represents any value of the underlying domain. It is represented in mongoDB by means of the string "$unknown".
- The *Undefined* type represents that it has no sense for the considered attribute to take any value of the domain, that is, no value is applicable. It is represented in mongoDB by means of the string "$undefined".
- A *Null* value represents that we cannot provide any value because we do not know whether it is applicable or not. To represent it, the BSON null type provided by mongoDB is used.

The fourth column in Table 1 illustrates the representation of each type in mongoDB by means of an example.

3.2 | Fuzzy relational operators

To perform queries on fuzzy data it is necessary to extend the classical relational operators. When a fuzzy relational operator is applied on two fuzzy numbers, it returns a value in $[0, 1]$ which represents the fulfilment degree (*cdeg*) for the considered fuzzy relational comparison. The calculation of this degree depends on the measure used. In our model, we use measures based on *possibility* and on *necessity*.

Our proposal provides support for the fuzzy extension of the classical relational operators, as Medina et al.[10] did in GEFRED: $=, >, <, \geqslant$ and $\leqslant$, using a possibility measure: *feq*, *fgt*, *flt*, *fgte* and *flte*, respectively, and using a necessity measure: *nfeq*, *nfgt*, *nflt*, *nfgte* and *nflte*, respectively.

Next, we show the definitions for these fuzzy operators when applied to trapezoidal distributions.

Let $A = [\alpha_A, \beta_A, \gamma_A, \delta_A]$ and $B = [\alpha_B, \beta_B, \gamma_B, \delta_B]$ be two trapezoidal possibility distributions defined on an ordered domain Ω; then, the compatibility degree obtained for each fuzzy comparison between them is calculated as shown in the following definitions:

Definition 1 (FEQ operator). The degree in which A and B are `possibly` equals is

$$feq(A, B) = \begin{cases} 1 & \text{if } \gamma_A \geq \beta_B \text{ and } \beta_A \leq \gamma_B, \\ 0 & \text{if } \delta_A \leq \alpha_B \text{ or } \alpha_A \geq \delta_B, \\ \dfrac{\delta_A - \alpha_B}{(\beta_B - \alpha_B) - (\gamma_A - \delta_A)} & \text{if } \delta_A > \alpha_B \text{ and } \gamma_A < \beta_B, \\ \dfrac{\delta_B - \alpha_A}{(\beta_A - \alpha_A) - (\gamma_B - \delta_B)} & \text{in another case.} \end{cases} \tag{2}$$

Definition 2 (NFEQ operator). The degree in which A and B are `necessarily` equals is

$$nfeq(A,B) = \begin{cases} 0 & \text{if } (\beta_A \le \alpha_B \text{ and } \alpha_A \ne \beta_B) \text{ or} \\ & (\gamma_A \ge \delta_B \text{ and } \delta_A \ne \gamma_B), \\ \min\left(\dfrac{\beta_A - \alpha_B}{(\beta_B - \alpha_B) - (\alpha_A - \beta_A)}, \right. & \text{if } \alpha_A < \beta_B \text{ and } \delta_A > \gamma_B, \\ \left. \dfrac{\gamma_A - \delta_B}{(\gamma_B - \delta_B) - (\delta_A - \gamma_A)}\right) & \\ \dfrac{\beta_A - \alpha_B}{(\beta_B - \alpha_B) - (\alpha_A - \beta_A)} & \text{if } \alpha_A < \beta_B \text{ and } \delta_A \le \gamma_B, \\ \dfrac{\gamma_A - \delta_B}{(\gamma_B - \delta_B) - (\delta_A - \gamma_A)} & \text{if } \alpha_A \ge \beta_B \text{ and } \delta_A > \gamma_B, \\ 1 & \text{in another case.} \end{cases} \tag{3}$$

Definition 3 (FGT operator). The degree in which A is `possibly` greater than B is

$$fgt(A,B) = \begin{cases} 1 & \text{if } \gamma_A \ge \delta_B, \\ \dfrac{\delta_A - \gamma_B}{(\delta_B - \gamma_B) - (\gamma_A - \delta_A)} & \text{if } \gamma_A < \delta_B \text{ and } \delta_A > \gamma_B, \\ 0 & \text{in another case.} \end{cases} \tag{4}$$

Definition 4 (NFGT operator). The degree in which A is `necessarily` greater than B is

$$nfgt(A,B) = \begin{cases} 1 & \text{if } \alpha_A \ge \delta_B, \\ \dfrac{\beta_A - \gamma_B}{(\delta_B - \gamma_B) - (\alpha_A - \beta_A)} & \text{if } \alpha_A < \delta_B \text{ and } \beta_A > \gamma_B, \\ 0 & \text{in another case.} \end{cases} \tag{5}$$

Definition 5 (FGTE operator). The degree in which A is `possibly` greater or equal than B is

$$fgte(A,B) = \begin{cases} 1 & \text{if } \gamma_A \ge \beta_B, \\ \dfrac{\delta_A - \alpha_B}{(\beta_B - \alpha_B) - (\gamma_A - \delta_A)} & \text{if } \gamma_A < \beta_B \text{ and } \delta_A > \alpha_B, \\ 0 & \text{in another case.} \end{cases} \tag{6}$$

Definition 6 (NFGTE operator). The degree in which A is `necessarily` greater or equal than B is

$$nfgte(A,B) = \begin{cases} 1 & \text{if } \alpha_A \geq \beta_B, \\ \dfrac{\beta_A - \alpha_B}{(\beta_B - \alpha_B) - (\alpha_A - \beta_A)} & \text{if } \alpha_A < \beta_B \text{ and } \beta_A > \alpha_B, \\ 0 & \text{in another case.} \end{cases} \tag{7}$$

Definition 7 (FLT operator)**.** The degree in which A is `possibly` less than B is

$$flt(A,B) = \begin{cases} 1 & \text{if } \beta_A \leq \alpha_B, \\ \dfrac{\alpha_A - \beta_B}{(\alpha_B - \beta_B) - (\beta_A - \alpha_A)} & \text{if } \beta_A > \alpha_B \text{ and } \alpha_A < \beta_B, \\ 0 & \text{in another case.} \end{cases} \tag{8}$$

Definition 8 (NFLT operator)**.** The degree in which A is `necessarily` less than B is

$$nflt(A,B) = \begin{cases} 1 & \text{if } \delta_A \leq \alpha_B, \\ \dfrac{\gamma_A - \beta_B}{(\alpha_B - \beta_B) - (\delta_A - \gamma_A)} & \text{if } \delta_A > \alpha_B \text{ and } \gamma_A < \beta_B, \\ 0 & \text{in another case.} \end{cases} \tag{9}$$

Definition 9 (FLTE operator)**.** The degree in which A is `possibly` less or equal than B is

$$flte(A,B) = \begin{cases} 1 & \text{if } \beta_A \leq \gamma_B, \\ \dfrac{\delta_B - \alpha_A}{(\beta_A - \alpha_A) - (\gamma_B - \delta_B)} & \text{if } \beta_A > \gamma_B \text{ and } \alpha_A < \delta_B, \\ 0 & \text{in another case.} \end{cases} \tag{10}$$

Definition 10 (NFLTE operator)**.** The degree in which A is `necessarily` less or equal than B is

$$nflte(A,B) = \begin{cases} 1 & \text{if } \delta_A \leq \gamma_B, \\ \dfrac{\gamma_A - \delta_B}{(\gamma_B - \delta_B) - (\delta_A - \gamma_A)} & \text{if } \delta_A > \gamma_B \text{ and } \gamma_A < \delta_B, \\ 0 & \text{in another case.} \end{cases} \tag{11}$$

3.3 | Logic connectives

To perform fuzzy queries that combine atomic fuzzy conditions, it is necessary the use of logic connectives. In the model that underlies our fuzzy extension of mongoDB, we use the *minimum* *t*-norm for the connective *AND*, and the *maximum t*-conorm for the connective *OR*. For the NOT operator we use the expression:

$$NOT(f(x)) = 1 - f(x). \tag{12}$$

3.4 | Flexible conditions and query optimisation

A fuzzy query on a database is performed by means of a combination of atomic flexible conditions. An atomic flexible condition implies the application of a fuzzy comparison operator on a set of database rows with a given threshold of fulfilment. For instance, if we consider the fuzzy set of rows that possibly satisfy the fuzzy condition C on the attribute A, its membership function can be defined as shown in Equation (13).

$$\Pi(C/r) = \sup_{d \in D(A)} (\Pi_{A(r)}(d), \mu_C(d)), \tag{13}$$

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 combined with a fulfilment threshold (T) is called atomic flexible condition and it filters the rows that satisfy the flexible condition exceeding this threshold; it is noted as $\langle C, T \rangle$.

When an atomic flexible condition is applied on trapezoidal fuzzy values, it is possible to apply an indexing strategy to enhance the performance of the fuzzy query. This strategy is based on the application of the indexing principle introduced in Reference [26]. In References [27–30], this principle was implemented using the indexing mechanisms available on classical RDBMS. The idea of the indexing principle consists of applying a *preselection* criterion that retrieves the rows that possibly satisfy the flexible condition for a given threshold. Therefore, the rows that do not satisfy the flexible condition are discarded. This *preselection* criterion allows using indexes built on the components of the trapezoidal fuzzy numbers to improve the retrieval performance. Later, it is necessary to filter again this set of rows to retrieve those ones that effectively satisfy the flexible condition exceeding this threshold.

To illustrate the use of the *preselection* criterion, we will consider an atomic flexible condition on a fuzzy attribute $A(t)$ to get the rows that are *equal* to a trapezoidal value C at a threshold T, using a *possibility measure*. Given the trapezoidal fuzzy attribute values represented as $\Pi_{A(t)} = [\alpha_{A(t)}, \beta_{A(t)}, \gamma_{A(t)}, \delta_{A(t)}]$ the *preselection* criterion is expressed as Equation (14) shows

$$ps'(C/t, T) \Leftrightarrow \delta_{A(t)} \geq l_{base(\langle C,T \rangle)} \wedge \alpha_{A(t)} \leq u_{base(\langle C,T \rangle)}, \tag{14}$$

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 Equation (15). For the 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.

$$base(\langle C, T \rangle) = \begin{cases} supp(C), & T = 0, \\ supp_T(C), & 0 < T \leq 1. \end{cases} \tag{15}$$

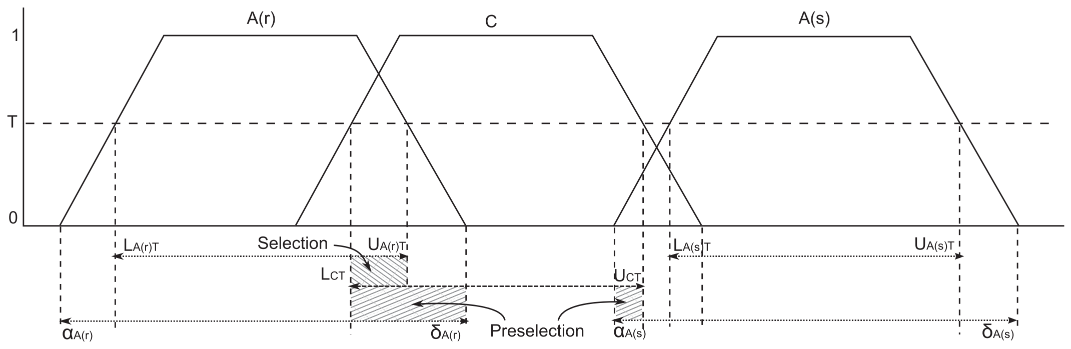


FIGURE 1 Example of the computation of the preselection and the selection sets

With the help of Figure 1 we will illustrate how to obtain the rows that satisfy the *preselection* criterion and those ones that satisfy the *selection* criterion. This figure shows two rows r, s on whose attribute A we apply a flexible condition $\langle C, T\rangle$ with $\mu_C = [\alpha, \beta, \gamma, \delta]$, $\Pi_{A(r)} = [\alpha_{A(r)}, \beta_{A(r)}, \gamma_{A(r)}, \delta_{A(r)}]$ and $\Pi_{A(s)} = [\alpha_{A(s)}, \beta_{A(s)}, \gamma_{A(s)}, \delta_{A(s)}]$.

As we can deduce from Figure 1, the $[L_{CT}, U_{CT}]$ limits are calculated by means of the next expressions:

$$\begin{aligned} L_{CT} &= T * \beta + (1 - T) * \alpha, \\ U_{CT} &= T * \gamma + (1 - T) * \delta. \end{aligned} \tag{16}$$

On the other hand, the limits for any tuple t in the attribute A ($L_{A(t)T}$ and $U_{A(t)T}$) are computed as follows:

$$\begin{aligned} L_{A(t)T} &= T * \beta_{A(t)} + (1 - T) * \alpha_{A(t)}, \\ U_{A(t)T} &= T * \gamma_{A(t)} + (1 - T) * \delta_{A(t)} \end{aligned} \tag{17}$$

Visually, we can check that $A(s)$ and $A(r)$ satisfy the *preselection* criterion (Equation 14). However, only $A(r)$ satisfies the *selection* criterion because it accomplishes the selection condition:

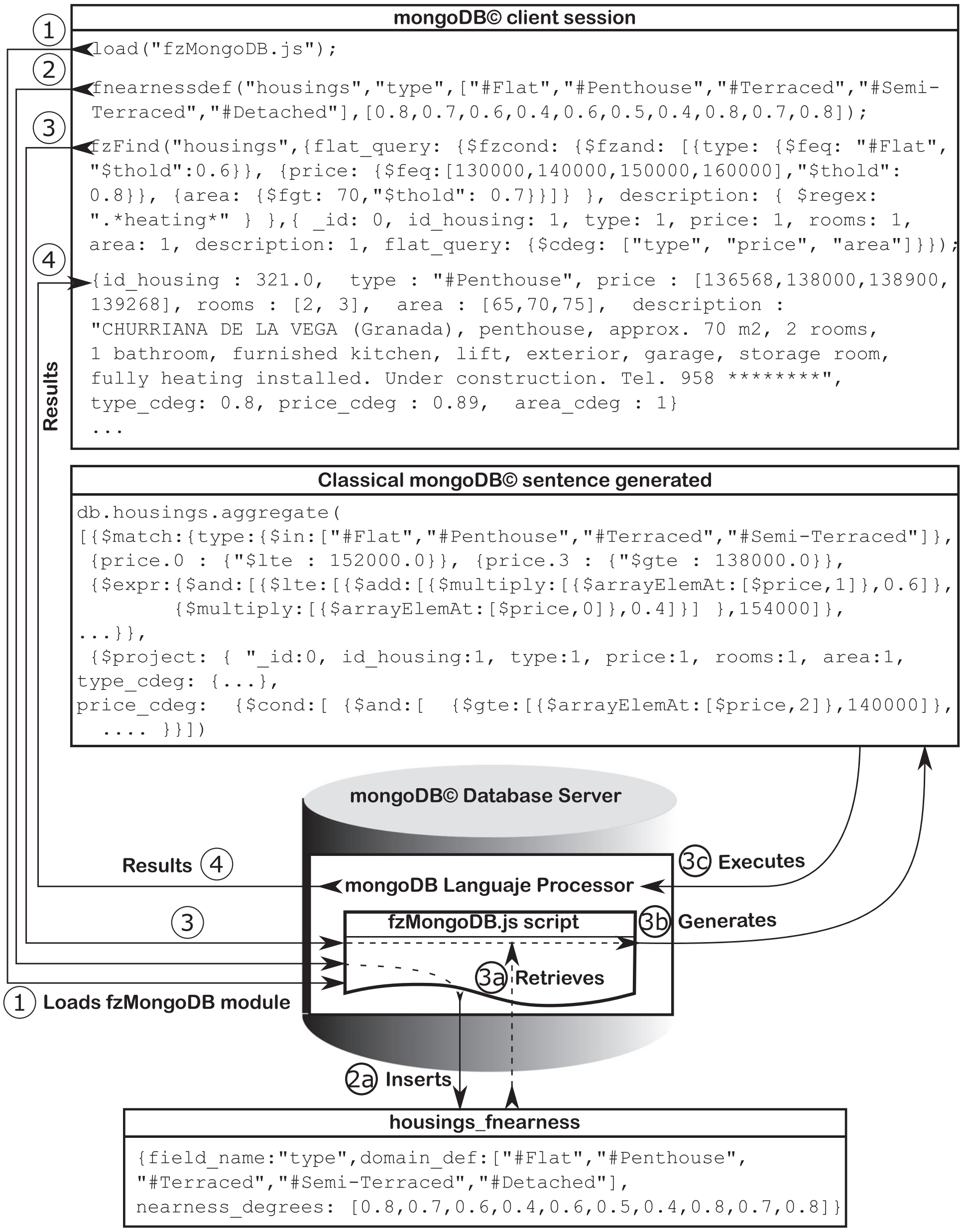
$$L_{A(r)T} \leq U_{CT} \wedge U_{A(r)T} \geq L_{CT}. \tag{18}$$

For the sake of simplicity, we will note as $\langle feq(C, A), T\rangle$ the illustrated atomic flexible condition.

Using a similar reasoning (see Reference [28]), we can obtain the expression to get the *preselection* and *selection* sets when applying an atomic flexible condition based on a necessity measure, $\langle nfeq(C, A), T\rangle$.

The preselection condition is given by

$$\beta_{A(t)} \geq L_{CT} \wedge \gamma_{A(t)} \leq U_{CT} \tag{19}$$



```
load("fzMongoDB.js");

fnearnessdef("housings","type",["#Flat","#Penthouse","#Terraced","#Semi-
Terraced","#Detached"],[0.8,0.7,0.6,0.4,0.6,0.5,0.4,0.8,0.7,0.8]);

fzFind("housings",{flat_query: {$fzcond: {$fzand: [{type: {$feq: "#Flat",
"$thold":0.6}}, {price: {$feq:[130000,140000,150000,160000],"$thold":
0.8}}, {area: {$fgt: 70,"$thold": 0.7}}]} }, description: { $regex:
".*heating*" } },{ _id: 0, id_housing: 1, type: 1, price: 1, rooms: 1,
area: 1, description: 1, flat_query: {$cdeg: ["type", "price", "area"]}});

{id_housing : 321.0,  type : "#Penthouse", price : [136568,138000,138900,
139268], rooms : [2, 3],  area : [65,70,75],  description :
"CHURRIANA DE LA VEGA (Granada), penthouse, approx. 70 m2, 2 rooms,
1 bathroom, furnished kitchen, lift, exterior, garage, storage room,
fully heating installed. Under construction. Tel. 958 ********",
type_cdeg: 0.8, price_cdeg : 0.89,  area_cdeg : 1}
...
```

```
db.housings.aggregate(
[{$match:{type:{$in:["#Flat","#Penthouse","#Terraced","#Semi-Terraced"]},
 {price.0 : {"$lte : 152000.0}}, {price.3 : {"$gte : 138000.0}},
 {$expr:{$and:[{$lte:[{$add:[{$multiply:[{$arrayElemAt:[$price,1]},0.6]},
        {$multiply:[{$arrayElemAt:[$price,0]},0.4]}]} },154000]},
...}},
 {$project: { "_id:0, id_housing:1, type:1, price:1, rooms:1, area:1,
type_cdeg: {...},
price_cdeg:  {$cond:[ {$and:[  {$gte:[{$arrayElemAt:[$price,2]},140000]},
  .... }}])
```

```
{field_name:"type",domain_def:["#Flat","#Penthouse",
"#Terraced","#Semi-Terraced","#Detached"],
nearness_degrees: [0.8,0.7,0.6,0.4,0.6,0.5,0.4,0.8,0.7,0.8]}
```

FIGURE 2 Example of the processing of a fuzzy query

and, the *selection* condition

$$L_{A(t)T} \geq L_{CT} \wedge U_{A(t)T} \leq U_{CT}, \tag{20}$$

where

$$\begin{aligned} L_{CT} &= T * \beta + (1 - T) * \alpha, \\ U_{CT} &= T * \gamma + (1 - T) * \delta \end{aligned} \tag{21}$$

and

$$L_{A(t)T} = T * \beta_{A(t)} + (1 - T) * \alpha_{A(t)},$$
$$U_{A(t)T} = T * \gamma_{A(t)} + (1 - T) * \delta_{A(t)}. \tag{22}$$

For the rest of the fuzzy comparison operators described in Section 3.2, the expressions to compute the *preselection* and *selection* sets are the following:

- $\langle fgt(C, A), T \rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\delta_{A(t)} \geq L_{CT}, \tag{23}$$

$$L_{A(t)T} \geq L_{CT}, \tag{24}$$

where

$$L_{CT} = T * \delta + (1 - T) * \gamma, \tag{25}$$

$$L_{A(t)T} = T * \gamma_{A(t)} + (1 - T) * \delta_{A(t)}. \tag{26}$$

- $\langle nfgt(C, A), T \rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\beta_{A(t)} \geq L_{CT}, \tag{27}$$

$$L_{A(t)T} \geq L_{CT}, \tag{28}$$

where

$$L_{CT} = T * \delta + (1 - T) * \gamma, \tag{29}$$

$$L_{A(t)T} = T * \alpha_{A(t)} + (1 - T) * \beta_{A(t)}. \tag{30}$$

- $\langle fgte(C, A), T \rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\delta_{A(t)} \geq L_{CT}, \tag{31}$$

$$L_{A(t)T} \geq L_{CT}, \tag{32}$$

where

$$L_{CT} = T * \beta + (1 - T) * \alpha, \tag{33}$$

$$L_{A(t)T} = T * \gamma_{A(t)} + (1 - T) * \delta_{A(t)}. \tag{34}$$

- $\langle nfgte(C, A), T\rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\beta_{A(t)} \geq L_{CT}, \tag{35}$$

$$L_{A(t)T} \geq L_{CT}, \tag{36}$$

where

$$L_{CT} = T * \beta + (1 - T) * \alpha, \tag{37}$$

$$L_{A(t)T} = T * \alpha_{A(t)} + (1 - T) * \beta_{A(t)}. \tag{38}$$

- $\langle flt(C, A), T\rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\alpha_{A(t)} \leq U_{CT}, \tag{39}$$

$$U_{A(t)T} \leq U_{CT}, \tag{40}$$

where

$$U_{CT} = T * \alpha + (1 - T) * \beta, \tag{41}$$

$$U_{A(t)T} = T * \beta_{A(t)} + (1 - T) * \alpha_{A(t)}. \tag{42}$$

- $\langle nflt(C, A), T\rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\alpha_{A(t)} \leq U_{CT}, \tag{43}$$

$$U_{A(t)T} \leq U_{CT}, \tag{44}$$

where

$$U_{CT} = T * \alpha + (1 - T) * \beta \tag{45}$$

$$U_{A(t)T} = T * \beta_{A(t)} + (1 - T) * \alpha_{A(t)} \tag{46}$$

- $\langle flte(C, A), T\rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\alpha_{A(t)} \leq U_{CT}, \tag{47}$$

$$U_{A(t)T} \leq U_{CT}, \tag{48}$$

where

$$U_{CT} = T * \gamma + (1 - T) * \delta, \tag{49}$$

$$U_{A(t)T} = T * \beta_{A(t)} + (1 - T) * \alpha_{A(t)} \tag{50}$$

- $\langle nflte(C, A), T\rangle$. The *preselection* and the *selection* condition are given, respectively, by next expressions:

$$\gamma_{A(t)} \leq U_{CT}, \tag{51}$$

$$U_{A(t)T} \leq U_{CT}, \tag{52}$$

where

$$U_{CT} = T * \gamma + (1 - T) * \delta, \tag{53}$$

$$U_{A(t)T} = T * \delta_{A(t)} + (1 - T) * \gamma_{A(t)}. \tag{54}$$

4 | SYNTAX OF THE FUZZY EXTENSION OF mongoDB

The fuzzy database model underlying this proposal is based on the fuzzy relational database model described in References [10,25]. The fuzzy data model, the fuzzy comparison operators and the fuzzy relational operators are adapted from this fuzzy relational model to the document-based database model of mongoDB.

Our proposal for a fuzzy extension of mongoDB is based on the fuzzy database model described in Section 3; in this section we will describe the syntax to operate with this fuzzy extension. All the components of this fuzzy syntax are extensions of the mongoDB language, so they are consistent with the syntax it provides. The main elements of the mongoDB syntax have been described in Section 2.

4.1 | Fuzzy data and fuzzy data definition statements

The proposed fuzzy extension of mongoDB uses the data model described in Section 3.1 with the data types summarised in Table 1.

To operate with the fuzzy extension of mongoDB, it is necessary to use several system collections that store metadata about the fuzzy data stored in the database. It is also necessary to define some functions to manipulate such metadata.

The *flabeldef* function is provided to define fuzzy linguistic labels and the syntax is

```
  flabeldef(<collection>,<field>,<flabel_name>,<fuzzy_
definition>)
```

being, `<flabel_name>` a *string* with the name of the label to be defined, `<collection>` and `<field>` *strings* identifying where is the label defined, `<fuzzy_definition>` a *fuzzy* *number* expressed in *crisp*, *interval*, *triangular* or *trapezoidal* format defining such linguistic label. The execution of this function returns a new document with the provided values and it is inserted in a collection named `<collection>_flabel`, according to the following format:

```
{
field_name: <field>,
label_name: <flabel_name>,
label_def: <fuzzy_definition>
}
```

If the label in this field is already defined, its definition is updated in the `<collection>_flabel` collection with the new `<fuzzy_definition>` value.

To delete a label, the function *flabeldel* is used in the way

```
flabeldel(<collection>,<field>,<flabel_name>)
```

To define a scalar domain on a field and the nearness relation defined on it, the function *fnearnessdef* is provided with this syntax:

```
fnearnessdef(<collection>,<field>,<scalars_array>,
<degrees_array>)
```

where `<collection>` and `<field>` determine where is the domain defined, `<scalars_array>` is an array of *Scalar* datatype values that defines the scalar domain, `<degrees_array>` is an array of numbers in $[0, 1]$ that provides the nearness degree between each pair of values of the domain. The content of the array of nearness degrees depends on the order in the `<scalars_array>` field, in this way: if `<scalars_array>` is given by `["#s1", "#s2",…, "#sn"]` then to define the nearness degree between each pair of scalars an `array_nearness` array must be provided in the form: `[<nd(#s1,#s2)>,<nd(#s1,#s3)>,…,<nd(#s1,#sn)>,<nd(#s2,#s3)>,…,<nd(#s2,#sn)>,…,<nd(#sn-1,#sn)>]`, where each `<nd(#si,#sj)>` represents the nearness degree between the scalar in the *i* position and the scalar in the *j* position within the `<scalars_array>`.

Calling this function results in the insertion of a new document (or the replacement of an existing one) in a metadata collection called `<collection>_fnearness`. The structure of this document is as follows:

```
{
field_name: <field>,
domain_def: <scalars_array>,
nearness_degrees: <degrees_array>
}
```

To remove the definition of a scalar domain and the nearness relation defined on it the function *fnearnessdel* is provided, with the format

```
fnearnessdel(<collection>,<field>)
```

4.2 | Fuzzy expressions

The expressions that can be used in mongoDB are extended to handle the supported fuzzy data types and to perform fuzzy operations on them. These fuzzy expressions can be combined with the "classical" mongoDB expressions.

4.2.1 | Fuzzy comparison expressions

The fuzzy extension provides fuzzy comparators (based on both possibility and necessity measures) that extend the classical relational operators for fuzzy comparisons. The basic syntax is

`{<numerical_field>: {<fuz_rel_op>: <fuz_value> [,<threshold_clause>]}}`

where `<fuz_rel_op>` represents one of the provided fuzzy extensions for the comparison operators (`$eq`, `$ne`, `$lt`, `$gt`, `$lte` and `$gte`) based on possibility measure: `$feq`, `$fne`,`$flt`,`$fgt`,`$flte` and `$fgte` or, the ones based on the necessity measure: `$nfeq`, `$nfne`,`$nflt`,`$nfgt`,`$nflte` and `$nfgte`. The `<numerical_field>` item can store values of any of the fuzzy data types described on Table 1, excluding the *Scalar* data type. Also, `<fuz_value>` can be of any of these data types, excluding a *Scalar* data type value. The optional `<threshold_clause>` establishes the threshold (in [0, 1]) that the fulfilment degree of the comparison must satisfy for the retrieval of a document. Its syntax is `$thold: <value_in_0_1>`; when not established, the threshold is set to 0.

The specific syntax for comparisons on *Scalar* fields is

`{<scalar_field>: {$feq: <scalar_value> [,<threshold_clause>]}}`

where the `<scalar_field>` can store values of the *Scalar*, *Undefined*, *Unknown* or *Null* data type (Table 1). If a document stores a value of a different type from the ones mentioned in this field, it is not retrieved. The `<scalar_value>` item can be of any of these data types. If a nearness relation is defined on the domain of this field then it will be used.

The following expressions are examples using the described syntax:

```
{price: {$lte: [1e5,1.2e,1.3e5,1.5e5]}}
{type: {$feq: "#Flat", $thold: 0.7}} //a nearness relation on the "type of housing" domain
     should be defined
{area: {$gte: [50,60,65]}}
{price: {$feq: "$Cheap"}} //The linguistic label "$Cheap" must be defined on the "price"
     domain
```

4.2.2 | Fuzzy logic expressions

The fuzzy extension of mongoDB provides the operators `$fzand`, `$fzor` and `$fznot`, which implement the fuzzy extensions for the logical operators: `$and`, `$or` and `$not`.

`$fzand` joins fuzzy expressions using the min *T*-norm and returns the documents satisfying all the fuzzy conditions over the imposed threshold, `$fzor` joins fuzzy expressions using the max *T*-conorm and returns the documents satisfying any fuzzy expression over the imposed threshold and `$fznot` inverts the effect of a query expression and returns documents that do not match the fuzzy expression at the established threshold. Their basic syntax is

```
{$fzand: [<fuzzy_expression_1>, <fuzzy_expression_2>,…]}
{$fzor: [<fuzzy_expression_1>, <fuzzy_expression_2>,…]}
{$fznot: <fuzzy_expression>}
```

4.2.3 | Fuzzy conditions

To express a fuzzy predicate by means of a fuzzy expression, the `$fzcond` operator is used. The syntax is

```
{<fzcond_name>: {$fzcond: <fuzzy_expression>}}
```

The `<fzcond_name>` assigned to the fuzzy predicate can be addressed in the `$cdeg` operator to show the fulfilment degree to which each document matches this fuzzy predicate, as described below. In a statement might appear several fuzzy predicates and, therefore, several `$fzcond` operators.

4.2.4 | Fuzzy projection expressions

The projection component of a fuzzy statement is used in the same way as in the classical statement, with the exception that the `$cdeg` operator can be included. As we previously mentioned, the `$cdeg` operator is used to return the compatibility degree for the whole fuzzy comparison or for each attribute in the provided array. The following sentences are valid syntax expressions for this operator:

```
<fzcond_name> : {$cdeg : 1} \\returns for each document, the compatibility degree for the
     whole fuzzy comparison
<fzcond_name> : {$cdeg: <attribute>} \\returns, for each document, the compatibility degree
      for the <attribute> of the fuzzy comparison
<fzcond_name> : {$cdeg: [<attribute1>,<attribute2>,...]} \\returns, for each document, the
     compatibility degree for each attribute of the fuzzy comparison listed in the array
```

4.3 | Fuzzy CRUD syntax

The fuzzy extension of mongoDB provides four functions that extend the capability of the basic mongoDB CRUD operations to deal with fuzzy data. These functions described below integrate the handling of classical data and operators along with the respective fuzzy ones.

For the *insert* command, or method, there is no need to provide a specific extension, because all fuzzy data types considered in Table 1 have a representation compatible with the classical types provided by mongoDB. In this way, the creation of new documents including instances of these fuzzy data types can be carried out with the provided *insert* command.

4.3.1 | FzFind

To extend the query capability to handle fuzzy predicates on fuzzy data, it is necessary to implement an extension of the *find* command. This extension, named *fzFind*, has the following syntax:

```
fzFind(<collection>,<filter>,<projection>,<options>)
```

where the string `<collection>` indicates the collection from which the retrieval is performed; `<filter>` is a document that expresses the query condition, which can combine *classical conditions* with fuzzy ones expressed as *fuzzy conditions* using the syntax shown in Section 4.2.3; `<projection>` is a document including which fields will be included in the result; in this document it is possible to combine *classical projection expressions* with *fuzzy projection expressions* following the syntax shown in Section 4.2.4; finally, the optional parameter `<options>` is a document composed of pairs option-value that allows one to pass the following optional options to the *fzFind* statement:

```
allowDiskUse: <boolean> // When set to true, enables writing to temporary files, it is for
      large output collections
explain: <boolean> // When set to true, outputs a document that describes the execution
     plan used by the server to execute the statement.
hint: <string or document> // Specifies the index to use for the query
cursor: {batchSize: <0 or positive integer>} // Specifies the initial batch size for the
     cursor
```

The result of the invocation of the *fzfind* function is a cursor to handle the resulting collection, which supports the following cursor methods: *cursor.hasNext()*, *cursor.next()*, *cursor.toArray()*, *cursor.forEach()*, *cursor.map()*, *cursor.objsLeftInBatch()*, *cursor.itcount()* and *cursor.pretty()*. The functionality and use of these methods is described in the mongoDB documentation.[23]

4.3.2 | FzUpdate

This function extends the capability of the *Update* command to apply filters including fuzzy predicates. It has the following syntax:

```
fzUpdate(<collection>,<updates>,<options>)
```

where the string `<collection>` denotes the collection to be updated, `<options>` is a document composed of option-value pairs to send the classical options supported by the command *Update* and, `<updates>` is an array with one or many update statements, with the format

```
[{q: <filter>,u: <document>,upsert: <boolean>,multi:
<boolean>,…},…]
```

where `<filter>` is a document to select the documents to be updated (the format is the same of the `fzFind` command) and `<document>` is the modified document. The behaviour of the parameters `upsert` and `multi` is the same as described for the update method in Section 2.2.

4.3.3 | FzDelete

To extend the capability of the *Delete* command to handle fuzzy predicates, this function is implemented according to the following format:

```
fzDelete(<collection>,<deletes>,<options>)
```

where the string `<collection>` denotes the collection from which documents will be removed, `<options>` is used to send the classical options supported by the command *Delete* and, `<deletes>` is an array with one or many delete statements, with the format

```
[{q: <filter>, limit: <integer>, collation: <document>}, ...]
```

where `<filter>` is a document to determine which documents will be deleted, `<limit>` specifies the number of matching documents to delete (**0** to delete all matching documents or **1** to delete only one) and, `collation` is a document to provide language-specific rules for string comparison.

5 | FzMongoDB MODULE IMPLEMENTATION

The built *fzMongoDB* module consists of a script written in JavaScript which, once loaded in a mongoDB session through the *load* command, provides mongoDB with the capacity for representing and handling fuzzy data. To build this module, we have followed a strategy based on the formulation of a syntax that is consistent with the classical syntax provided by mongoDB, as described in Section 4. To implement this extended syntax, we have used some resources provided by the mongoDB platform, which are described in Section 2, to translate the fuzzy mongoDB statements into classical mongoDB statements whose direct execution provides the expected fuzzy results.

To execute the *fzFind* statement, *fzMongoDB* script uses the mongoDB method *aggregate*. The language processor analyses the statement syntax with its parameters and generates the stages of the aggregation pipeline. From the *filter* parameter of this statement and invoking several internal functions, the language processor generates a mongoDB expression for the *$match* stage of the aggregation pipeline. In the same way, the processor generates a mongoDB expression for the *$project* stage from the *projection* parameter of the *fzFind* statement. Then, the method *aggregate* with the generated parameters is executed and the results are obtained.

For the *fzUpdate* and *fzDelete* statements, the *fzMongoDB* script uses the mongoDB method *runCommand*, generating the parameters and options for this method from the input parameters of such statements.

For the *flabeldef*, *fnearnessdef*, *flabeldel* and *fnearnessdel* statements, the *fzMongoDB* script uses the mongoDB methods *createCollection*, *insert*, *update* and *remove*, generating the parameters and options for these methods from the input parameters of such statements.

5.1 | An example of FzMongoBD module processing

With the assistance of Figure 2, we will illustrate the functioning of the *fzMongoDB* module focusing on the most complex fuzzy statement, the *fzFind* statement. To do it, we will use the collection of estates denominated *housings* used in Section 2.2, where each document describes the characteristics of a property, some of them by means of fuzzy values. To show the structure of each document we will use an *insert* command to insert an example of a property:

```
db.housings.insert({
    id_housing: 321,
    type: "#Penthouse", //A scalar value with a nearness relation defined below
    price: [136568,138000,138900,139268], //Trapezoidal distribution
    rooms: [2, 3], //Interval value
    area: [65,70,75], //Triangular value
    description: "CHURRIANA DE LA VEGA (Granada), penthouse, approx. 70 m2, 2 rooms, 1
         bathroom, furnished kitchen, lift, exterior, garage, storage room, fully heating
         installed. Under construction. Tel. 958 ********"
})
```

First, the *fzMongoDB* script is loaded (step 1 in Figure 2) through a classical client session connected to the mongoDB server. To show how a fuzzy query on the housings collection is processed, it is necessary to define the underlying domain for the field `type` and a nearness relation on the values of this domain. According to the syntax shown in Section 4.1, we will execute the next statement (step 2 in Figure 2):

```
fnearnessdef("housings","type",["#Flat","#Penthouse","#Terraced","#Semi-Terraced","#
    Detached"],[0.8,0.7,0.6,0.4,0.6,0.5,0.4,0.8,0.7,0.8])
```

As the result of the execution of this statement, the collection named *housings_fnearness* is created with the document shown in step 2a of Figure 2.

Now, let us suppose we want to retrieve “flats” with a price of approximately 145,000 euros but not more than 160,000 nor less than 130,000, with an area upper than 70 m^2, with a threshold of 0.6 for the type of house, 0.8 for the price and, 0.7 for the area, showing the relevant information of the estates satisfying the query. Using the syntax described in Section 4.3.1, this query can be written in the client session (step 3 in Figure 2) as follows:

```
fzFind("housings",
{//filter
 flat_query:
  {$fzcond:
    {$fzand:
      [
      {type: {$feq: "#Flat", "$thold": 0.6}},
      {price: {$feq:[130000,140000,150000,160000],"$thold": 0.8}},
      {area:{$fgt: 70,"$thold": 0.7}}
      ]
    }
  },
 description: { $regex: ".*heating*" } //Classical condition
},
{//projection
 _id: 0, id_housing: 1, type: 1, price: 1, rooms: 1, area: 1,
 description: 1, flat_query: {$cdeg: ["type", "price", "area"]}
})
```

When this statement with these parameters is sent and is processed by the *fzMongoDB* script, it generates (step 3b in Figure 2) the following mongoDB code:

```
db.housings.aggregate(
  [
   {$match: { <query> }}, {$project: { <specifications> }}
  ]
)
```

where `<query>` is an expression which translates the fuzzy and classic predicates of the clause `<filter>` of the `fzFind` command, to a directly executable classic mongoDB® expression. For this example, the structure of this expression is

```
  <query>:<flat_predicate_translation>+","+<price_predicate_translation>+
  ","+<area_predicate_translation>
```

where for instance, the code for the two first sub expressions is

```
 <flat_predicate_translation>:
   type:{ $in:["#Flat","#Penthouse","#Terraced","#Semi-Terraced"]}

 <price_predicate_translation>:
   {price.0 : {"$lte : 152000.0}}, //pre-selection condition a)
   {price.3 : {"$gte : 138000.0}}, //pre-selection condition b)



     $expr:{ //Selection condition
      $and:[
       {$lte:[
         {$add:[
           {$multiply:[{$arrayElemAt:[$price,1]},0.6]},
           {$multiply:[{$arrayElemAt:[$price,0]},0.4]}
              ]
         },154000
             ]
       },
       {$gte:[
         {$add:[
           {$multiply:[{$arrayElemAt:[$price,2]},0.6]},
           {$multiply:[{$arrayElemAt:[$price,3]},0.4]}
              ]
         },136000
             ]
       }
           ]
           }
```

Note that, if we have created the following indexes on the *price* field:

```
db.housings.createIndex({price.0: 1, price.3: 1})
db.housings.createIndex({price.3: 1, price.0: 1})
```

mongoDB applies the pre-preselection condition (shown in the two first lines of the previous code) using the most efficient index. This precondition filters those properties that finally fulfil the condition imposed on the price, speeding up the recovery process.

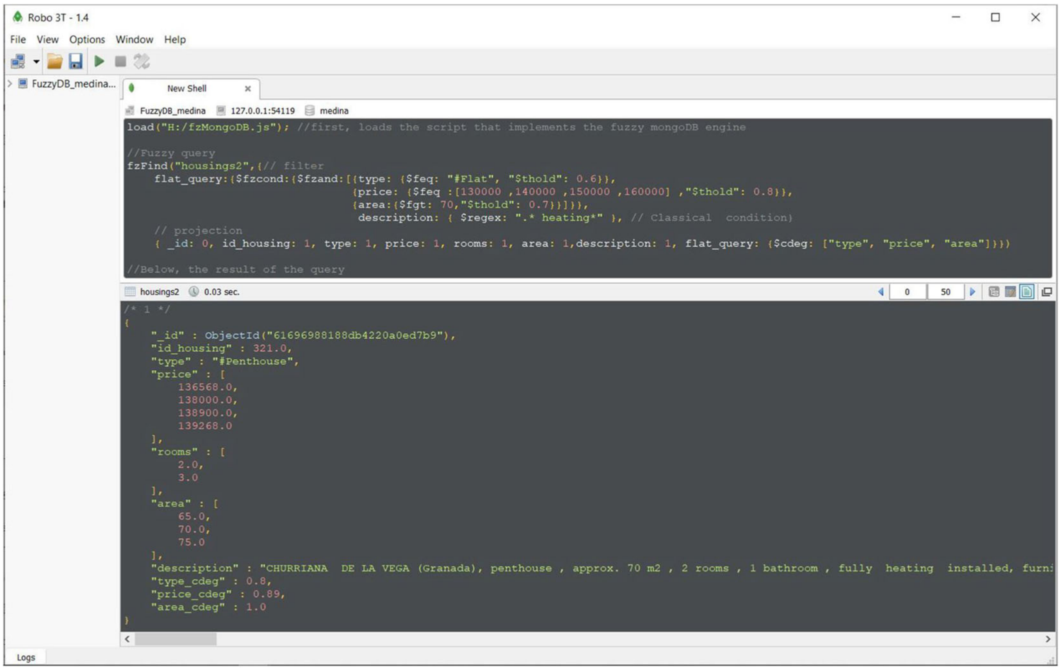
Finally, `<specifications>` is an expression which translates the `<projection>` clause of the `fzFind` command into a classical mongoDB code; by means of this code, the information shown as result of the query (including compatibility degrees for the fuzzy predicates) is generated. For this example, the expression is

```
<specifications>:
"_id:0, id_housing:1, type:1, price:1, rooms:1, area:1,+
<type_cdeg_code>+",+<price_cdeg_code>+",+<area_cdeg_code>
```

where for instance, the code for the second and third sub expressions is

```
<type_cdeg_code>:
  type_cdeg:
  {$switch: {
          branches: [
                     {case: {type:"#Flat"}, then: 1},
                     {case: {type:"#Penthouse"}, then: 0.8},
                     {case: {type:"#Terraced"}, then: 0.7},
                     {case: {type:"#Semi-Terraced"}, then: 0.6}
                    ],
          default: 0
           }
   }

<price_cdeg_code>: price_cdeg:
  {$cond:[ //Cond 1
    {$and:[ //if the below is true (the cores overlap)
      {$gte:[{$arrayElemAt:[$price,2]},140000]},
      {$lte:[{$arrayElemAt:[$price,1]},150000]}
          ]},1, //then retrieves 1
         {$cond:[ //Else Cond 2 (the supports do not overlap)
          {$or:[ //if the below is true
            {$lte:[{$arrayElemAt:[$price,3]},130000]},
            {$gte:[{$arrayElemAt:[$price,0]},160000]}
            ]},0, //Then retrieves 0
         {$cond:[ //Else Cond 3,
           {$and:[ //if the below is true (the upper alpha of the database value is
                greater than the lower alpha of the query)
             {$gt:[{$arrayElemAt:[$price,3]},130000]},
             {$lt:[{$arrayElemAt:[$price,2]},140000]}
                 ]}, //Them retrieves the calculation below
                    {$divide:[{$subtract:[{$arrayElemAt:[$price,3]},130000]},{$subtract
                         :[{$subtract:[140000,130000]},{$subtract:[{$arrayElemAt:[$price
                         ,2]},{$arrayElemAt:[$price,3]}]}]}]}
                    ]}, //Else (the lower alpha of the database value is lower than the
                          upper alpha of the query) retrieves the calculation below
                        {$divide:[{$subtract:[160000,{$arrayElemAt:[$price,0]}]},{
                            $subtract:[{$subtract:[{$arrayElemAt:[$price,1]},{
                            $arrayElemAt:[$price,0]}]},{$subtract:[150000,160000]}]}
                        ]}
         ]} //End Cond 3
      ]} //End Cond 2
 ]} //End Cond 1
```



Robo 3T - 1.4
File View Options Window Help
FuzzyDB_medina...
New Shell
FuzzyDB_medina 127.0.0.1:54119 medina
housings2 0.03 sec.
0
50
Logs

```
load("H:/fzMongoDB.js"); //first, loads the script that implements the fuzzy mongoDB engine

//Fuzzy query
fzFind("housings2",{// filter
    flat_query:{$fzcond:{$fzand:[{type: {$feq: "#Flat", "$thold": 0.6}},
                                 {price: {$feq :[130000 ,140000 ,150000 ,160000] ,"$thold": 0.8}},
                                 {area:{$fgt: 70,"$thold": 0.7}}]}},
                                  description: { $regex: ".* heating*" }, // Classical  condition}
    // projection
    { _id: 0, id_housing: 1, type: 1, price: 1, rooms: 1, area: 1,description: 1, flat_query: {$cdeg: ["type", "price", "area"]}})

//Below, the result of the query
```

```
/* 1 */
{
    "_id" : ObjectId("61696988188db4220a0ed7b9"),
    "id_housing" : 321.0,
    "type" : "#Penthouse",
    "price" : [
        136568.0,
        138000.0,
        138900.0,
        139268.0
    ],
    "rooms" : [
        2.0,
        3.0
    ],
    "area" : [
        65.0,
        70.0,
        75.0
    ],
    "description" : "CHURRIANA  DE LA VEGA (Granada), penthouse , approx. 70 m2 , 2 rooms , 1 bathroom , fully  heating  installed, furni
    "type_cdeg" : 0.8,
    "price_cdeg" : 0.89,
    "area_cdeg" : 1.0
}
```

FIGURE 3 Use of the *fzMongoDB* module through of a Robo 3T client [Color figure can be viewed at wileyonlinelibrary.com]

Then, this classical code generated by the *fzMongoDB* script is automatically sent to the mongoDB language processor to be executed (step 3c in Figure 2). As result, the mongoDB server returns to the mongoDB client, the list of documents (housings) that satisfy the fuzzy query (step 4 in Figure 2).

Figure 3 shows the execution of the query described in Figure 2 through a classical mongoDB client, Robo 3T in this case. Once the *fzMongoDB* script is loaded into a mongoDB classical client session, all supported fuzzy statements are available to the user.

6 | PERFORMANCE EVALUATION

The fuzzy extension module implements the optimisation described in Section 3.4 to improve the performance of the retrieval operations, by means of the use of indexes in the query execution plan. We have designed two sets of experiments to evaluate the performance enhancement of queries based on possibility and on necessity. The experimental setups have been developed on a mongoDB database server ver. 4.2.11 running on a server having a Core i7 CPU with 4 cores running at 3.4 GHz, 32 GB of RAM and a 512 GB SSD as the secondary storage.

6.1 | Indexing performance on queries based on possibility

To evaluate the performance impact of our indexing strategy for queries based on possibility, we have generated a set of nine collections and a set of 37,714 queries using the *feq* operator.

These queries have been executed using fulfilment thresholds in $[0, 1]$. To focus on the performance of the indexing strategy, the designed queries only retrieve the amount of documents that satisfy the fuzzy query.

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

- Three collections with 10^5 documents, three collections with 10^6 documents, and three collections with 10^7 documents. The documents include a field called "trape" that stores an array that represents each trapezoidal value generated. For each collection size, there are three variants, which store 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.
- In each collection, two compound indexes are created, one on the *alpha* and *delta* values of the "trape" field, and the other on the *delta* and *alpha* values of this field.
- The set of queries is randomly generated using the same parameters as the documents generated for the database set.

Figure 4 shows the results of the tests performed to evaluate the performance of the possibility-based queries. To compare the performance enhancement of our indexing strategy, each query has been also executed performing a full scan of the collection (notated as *FS* in the figures). In this figure, the results of the queries executed using our indexing strategy are noted as *IDX*.

Figure 4A shows the average execution time of the queries (*AET*) with respect to the collection size. For collections of 10^5 documents the *AET* of the indexed queries (*IDX*) is about a half with respect to the execution based on full scan (*FS*); for collections with size 10^6 and 10^7 the *AET* of *IDX* is about four times less than *FS*.

In classical databases, the use of indexes is more efficient for queries that retrieve fewer tuples, that is, when the selectivity is high. Figure 4B–D shows the *AET* with respect to the selectivity of the query for collections of sizes 10^5, 10^6 and 10^7, respectively. As we can see, in general, the *AET* increases as selectivity increases. When we compare *IDX* performance with *FS*, we notice that the trend is, the larger the collection size, the better the performance of *IDX* with respect to *FS*, and the lower the selectivity, the better the *IDX* performance with respect to *FS*.

Figure 4E shows how the compliance threshold of the query affects the performance. When this threshold is set to 1, the *preselected* documents match with those finally selected, and it is not necessarily an additional filtering. Because of this, we can see that the performance increases as the threshold does. On the other hand, *IDX* performs about 10 times better than *FS* on average.

The use of the compound index (alpha, delta) is more efficient when the upper *alpha-cut* (U_{CT}) value of the queried trapeziod is low. This is so because predictably there will be fewer trapezoids in the database whose *alpha* value be less than this upper *alpha-cut* value. On the other hand, if the U_{CT} value is high the compound index (alpha, delta) will be more efficient. For intermediate values, the use of any of these indexes will be less efficient. Figure 4F shows this trend for *IDX* queries, also shows a better performance of *IDX* with respect to *FS* in any case.

6.2 | Indexing performance on queries based on necessity

To evaluate the performance of our indexing strategy on queries based on necessity, we have designed the following experimental setup:

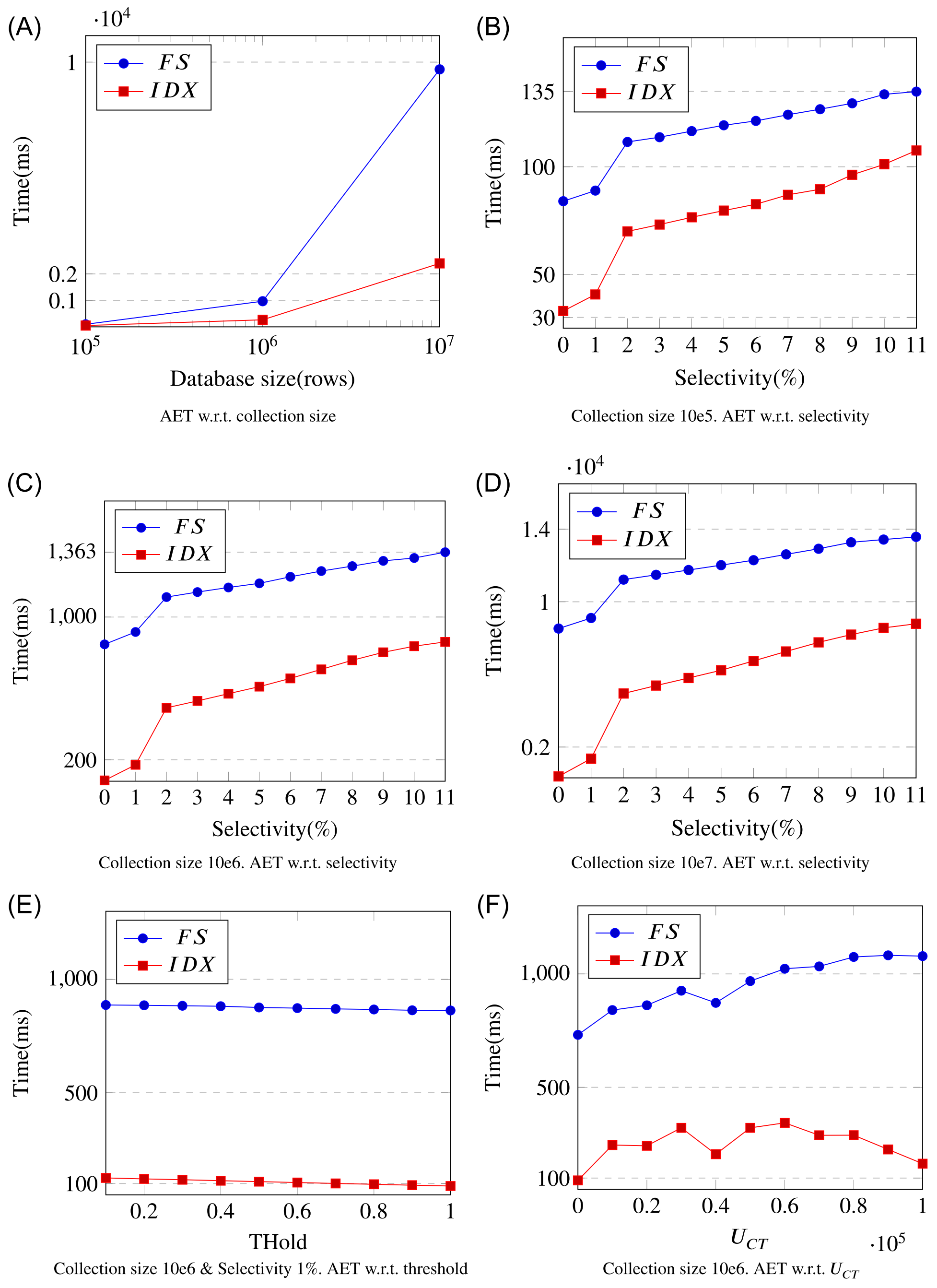


FIGURE 4 DBSet1. Performance tests on possibility based queries. AET, time of the queries; FS, full scan; IDX, indexing queries [Color figure can be viewed at wileyonlinelibrary.com]

- Five collections with 10^5, $3 \cdot 10^5$, 10^6, $5 \cdot 10^6$ and 10^7 documents, respectively. The documents of these collections contain a field that stores trapezoids with a fixed support size of 100, uniformly distributed on the domain ([0, 100000]). To generate the kernel of each trapezoid a uniform distribution has also been used.

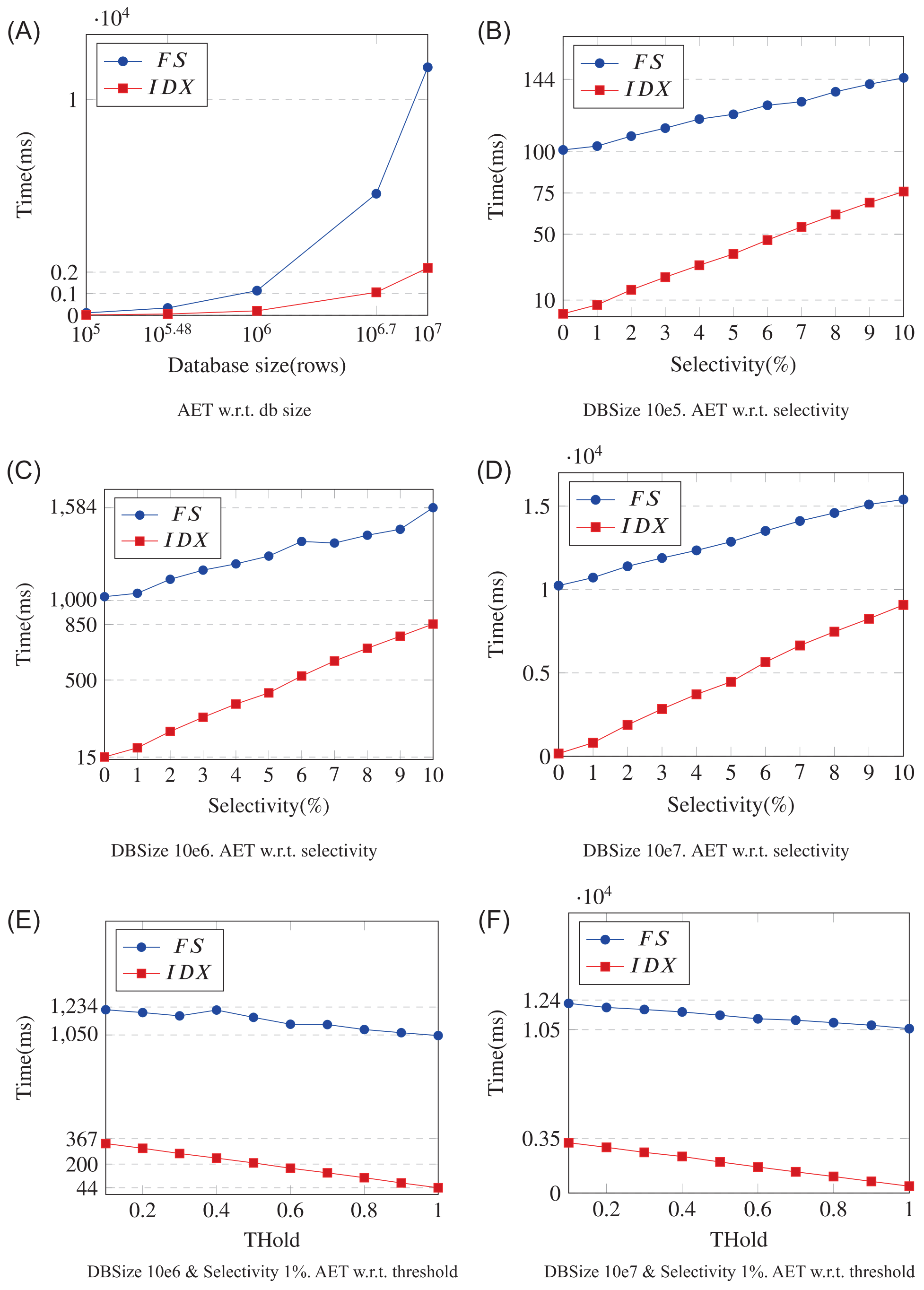


FIGURE 5 DBSet2. Performance tests on necessity-based queries. AET, time of the queries; FS, full scan; IDX, indexing queries [Color figure can be viewed at wileyonlinelibrary.com]

- A set of 40,000 queries was randomly generated using the same parameters as the documents generated for the database set.
- For each collection, a compound index on the beta and gamma values of the trapezoidal field has been created.

Figure 5 shows the results of the execution of the tests. Again, *IDX* represents the execution of the queries using the index, and *FS* the execution without any index.

Figure 5A shows the average execution time for all queries with respect to the collection size. As we see, the average performance of the queries using index (*IDX*) is about five times better than *FS*.

Figure 5B–D shows the average performance with respect to the selectivity of the query for collections with 100,000, 10^6 and 10^7 documents, respectively.

As we see in Figures 5E,F, the threshold of the query influences the performance as expected, the higher the threshold, the better the performance. Moreover, the average execution time for queries using *IDX* goes from a quarter of the time spent by *FS* at threshold 0 to the 24th part at threshold 1.

As all performed tests show, the use of our indexing strategy noticeably decreases the time expended in the execution of the queries with respect to its execution without the use of any index.

7 | CONCLUSION

We have proposed *fzMongoDB*, a fuzzy extension module for mongoDB database to provide it with the capability of handling flexible information, with four basic features:

1. It enables mongoDB database to represent and handle a wide variety of imprecise data types, and to perform queries on them based on possibility, necessity and similarity.
2. It provides a syntax extension which is consistent with the mongoDB syntax, allowing the natural integration of fuzzy clauses with classical ones.
3. It includes optimisations, based on indexing, to accelerate the retrieval process for queries based on possibility and on necessity. The performance tests executed demonstrate that the use of these indexing techniques speeds up the retrieval of documents, being better the performance as the collections augment the size.
4. It is implemented by means of a JavaScript script that translates the fuzzy statement into host statements of MongoBD, whose direct execution returns the results in the desired format. This means that, to exploit the fuzzy handling provided by this module, the user only needs to load the script into a classical mongoDB session and execute the sentences using the *fzMongoDB* syntax.

Our *fzMongoDB* module beats other proposals available in the literature like[19,20,22] which do not allow one to represent fuzzy data into the mongoDB database. Although the proposal in Reference [21] allows one to represent linguistic labels into mongoDB, it uses an inefficient method to query them. Besides, all of them use an external module to translate the fuzzy statements into the statements processable by the mongoDB database. In our case, the *fzMongoDB* module is executed into the mongoDB database. Also our *fzMongoDB* module is more detailed in the description of the fuzzy syntax, implementation and functioning than the other proposals. Further, it includes enhancements to accelerate the retrieval process.

Regarding future works, we are going to explore the possibility to extend the capability of the aggregation commands of mongoDB, such as `mapReduce` and `aggregate`, to deal with fuzzy information. This will allow one to take advantage of the processing scalability of this NoSQL database, integrating the capability for handling fuzzy information. MongoDB supports the representation of geospatial data through the GeoJSON objects and to express geospatial queries on them. It would be also interesting to extend its capacity to perform flexible geospatial queries.

ACKNOWLEDGEMENTS

This study has been partially supported by the MCIN/AEI/10.13039/501100011033 and FEDER: "Una manera de hacer Europa" under project PGC2018-096156-B-I00: *Recuperación y Descripción* de Imágenes mediante Lenguaje Natural usando Técnicas de Aprendizaje Profundo y Computación Flexible.

ORCID

Juan Miguel Medina https://orcid.org/0000-0002-0964-7324
Ignacio J. Blanco https://orcid.org/0000-0002-7825-9093
Olga Pons https://orcid.org/0000-0002-0149-0377

REFERENCES

1. Fernández A, Carmona CJ, Jesus dMJ, Herrera F. A view on fuzzy systems for big data: progress and opportunities. *Int J Comput Intell Syst*. 2016;9(Suppl 1):69-80. doi:10.1080/18756891.2016.1180820
2. Ducange P. Fuzzy models for big data mining. In: Fullér R, Giove S, Masulli F, eds. *Fuzzy logic and applications*. WILF, Springer International Publishing; 2019:257-260.
3. Smits G, Pivert O, Yager RR, Nerzic P. A soft computing approach to big data summarization. *Fuzzy Sets Syst*. 2018;348:4-20. doi:10.1016/j.fss.2018.02.017
4. Nayak A, Poriya A, Poojary D. Type of NoSQL databases and its comparison with relational databases. *Int J Appl Inf Syst*. 2013;5(4):16-19.
5. MongoDB. MongoDB DBMS. https://docs.mongodb.com/
6. Bosc P, Pivert O. SQLf: a relational database language for fuzzy querying. *IEEE Trans Fuzzy Syst*. 1995;3(1): 1-17. doi:10.1109/91.366566
7. Kacprzyk J, Zadrozny S. FQUERY for access: fuzzy querying for a Windows-based DBMS. In: Bosc P, Kacprzyk J, eds. *Fuzziness in Database Management Systems. Studies in Fuzziness*. Vol 5. Physica-Verlag HD; 1995:415-433.
8. Ma Z, Yan L. Generalization of strategies for fuzzy query translation in classical relational databases. *Inf Software Technol*. 2007;49(2):172-180. doi:10.1016/j.infsof.2006.05.002
9. Prade H, Testemale C. Generalizing database relational algebra for the treatment of incomplete or uncertain information and vague queries. *Inf Sci*. 1984;34(2):115-143. doi:10.1016/0020-0255(84)90020-3
10. Medina JM, Pons O, Vila MA. GEDRED: a generalized model of fuzzy relational databases. *Inf Sci*. 1994; 76(1):87-109. doi:10.1016/0020-0255(94)90069-8
11. Galindo J, Medina J, Pons O, Cubero J. A server for Fuzzy SQL queries. In: Andreasen T, Christiansen H, Larsen H, eds. *Flexible Query Answering Systems. Lecture Notes in Computer Science*. Vol 1495. Springer; 1998:164-174.
12. Shukla P, Darbari M, Singh V, Tripathi S. A survey of fuzzy techniques in object oriented databases. *Int J Sci Eng Res*. 2011;2(11):1-11. doi:10.1016/0306-4379(89)90017-3
13. Galindo J., ed. *Handbook of Research on Fuzzy Information Processing in Databases*. Information Science Reference; 2008.
14. Barranco CD, Campaña JR, Medina JM. Towards a fuzzy object-relational database model. In: Galindo J, ed. *Handbook of Research on Fuzzy Information Processing in Databases*. IGI Global; 2008:435-461.
15. Kacprzyk J, Zadrożny S, Tré GD. Fuzziness in database management systems: half a century of developments and future prospects. *Fuzzy Sets Syst*. 2015;281:300-307. Special Issue Celebrating the 50th Anniversary of Fuzzy Sets. doi:10.1016/j.fss.2015.06.011
16. Liu J, Zhang X. Modeling fuzzy relational database in HBase. *J Intell Fuzzy Syst*. 2016;31:1845-1857. doi:10.3233/JIFS-15899
17. Castelltort A, Martin T. Handling scalable approximate queries over NoSQL graph databases: Cypherf and the Fuzzy4S framework. *Fuzzy Sets Syst*. 2018;348:21-49. doi:10.1016/j.fss.2017.08.002

18. DB-Ranking. DB-Engines Ranking. https://db-engines.com/en/ranking
19. Abir BK, Amel GT. Towards fuzzy querying of NoSQL document-oriented databases. In: Laux, F. et al., eds. *Proceedings of the DBKDA 2015: The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications*, Rome, Italy, 24–29 May 2015. International Academy, Research, and Industry Association (IARIA):153–158.
20. Mehrab F, Harounabadi A. Apply uncertainty in document-oriented database (MongoDB) using F-XML. *J Adv Comput Res*. 2018;9:87-101.
21. Astachova IF, Samoilov NK, Kiseleva EI. Fuzzy request handler for Mongo QL derived from SQL. *J Phys: Conf Ser*. 2020;1479:012017. doi:10.1088/1742-6596/1479/1/012017
22. Fosci P, Psaila G. Towards flexible retrieval, integration and analysis of JSON data sets through fuzzy sets: a case study. *Information*. 2021;12(7):258. doi:10.3390/info12070258
23. MongoDB-Doc. MongoDB Documentation. https://docs.mongodb.com/manual/
24. BSON-Specification. BSON Specification. http://bsonspec.org/
25. Medina JM, Vila MA, Cubero JC, Pons O. Towards the implementation of a generalized fuzzy relational database model. *Fuzzy Sets Syst*. 1995;75:273-289.
26. Bosc P, Galibourg M. Indexing principles for a fuzzy data base. *Inf Syst*. 1989;14(6):493-499. http://www.sciencedirect.com/science/article/B6V0G-48TD2GC-HY/2/0b43ae6709c19a57591bccd54ad7386b
27. Medina JM, Barranco CD, Pons O. Evaluation of indexing strategies for possibilistic queries based on indexing techniques available in traditional RDBMS. *Int J Intell Syst*. 2016;31(12):1135-1165. doi:10.1002/int.21820
28. Medina JM, Barranco C, Pons O. Indexing techniques to improve the performance of necessity-based fuzzy queries using classical indexing of RDBMS. *Fuzzy Sets Syst*. 2018;351:90-107. doi:10.1016/j.fss.2017.09.008
29. Medina JM, Barranco CD, Pons O, Sanchez D. Building and Evaluation of Indexes for Possibilistic Queries on a Fuzzy Object-relational Database Management System. IEEE; 2017:1-6.
30. Medina JM, Barranco CD, Pons O. Indexes for necessity queries. Implementation and performance evaluation on a fuzzy object-relational database management system. In: *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE; 2018:1-6. doi:10.1109/FUZZ-IEEE.2018.8491608

How to cite this article: Medina JM, Blanco IJ, Pons O. A fuzzy database engine for mongoDB. *Int J Intell Syst*. 2022;1-34. doi:10.1002/int.22807