Processing Fuzzy Relational Queries Using Fuzzy Views

Emmanuel Doumard, Olivier Pivert, Grégory Smits, Virginie Thion

HAL Id: hal-02116133

https://hal.inria.fr/hal-02116133

Submitted on 30 Apr 2019

experimentation aimed to compare the performances of the two solutions described before. The experimental results are discussed and the pros and cons of each method are pointed out. Finally, Section VI recalls the main contributions of the paper and outlines perspectives for future research.

II. CONTEXT AND RELATED WORK

In this section, we recall some useful notions concerning fuzzy queries in a relational database context, with a particular focus on the SQLf language [5] which constitutes the framework considered in this work. We first deal with syntactic aspects, then we briefly discuss query-processing-related issues.

A. Refresher about Fuzzy Relational Queries

Dealing with fuzzy queries in a relational database context first implies to extend the classical notion of a relation into that of a *fuzzy relation*. A fuzzy relation is associated with a fuzzy concept ψ and is simply a relation where every tuple is assigned a membership degree in the unit interval, that reflects the extent to which the tuple satisfies ψ. The operations from the relational algebra can be extended to fuzzy relations along two lines: first, by considering fuzzy relations as fuzzy sets; second by introducing gradual predicates in the appropriate operations (selections and joins especially). The definitions of the extended relational operators can be found in [1]. As an illustration, we give the definition of the fuzzy selection operator hereafter:

$$\mu_{select(r,\,\varphi)}(t) = \top(\mu_r(t),\,\mu_\varphi(t))$$

where r denote a fuzzy relation, φ is a fuzzy predicate and $\top$ is a triangular norm (most usually, $\min$ is used).

Relational algebra is obviously too abstruse for non-experts to use, this is why no commercial DBMS offers it as a query language. More user-oriented languages (based on relational algebra) had to be defined, and the most famous one is of course SQL, in which queries are expressed by means of "blocks" involving clauses such as *select*, *from*, *where*, etc. The language called SQLf, whose initial version was presented in [5], extends SQL so as to support fuzzy queries. The general principle consists in introducing gradual predicates wherever it makes sense. The three clauses *select*, *from* and *where* of the base block of SQL are kept in SQLf and the *from* clause remains unchanged (except when fuzzy joins are used, in which

Abstract—**This paper proposes two original approaches to the processing of fuzzy queries in a relational database context. The general idea is to use views, either materialized or not. In the first case, materialized views are used to store the satisfaction degrees related to user-defined fuzzy predicates, instead of calculating them at runtime by means of user functions embedded in the query (which induces an important overhead). In the second case, abstract views are used to efficiently access the tuples that belong to the α-cut of the query result, by means of a derived Boolean selection condition.**

I. INTRODUCTION

The idea of making database management systems more flexible by switching from Boolean logic to fuzzy logic for interpreting queries is already quite old, as the first work on this topic dates back to the mid 70's. Indeed, the first paper considering fuzzy relational queries was authored by V. Tahani — then a Ph.D. student supervised by L.A. Zadeh — in 1977 [16]. Since that time, a great amount of work has of course been done, both for defining highly expressive fuzzy query languages (see for instance [12], [5], [1], [14]) and for devising efficient processing techniques suitable for such queries (see e.g. [4], [6], [17], [15]). As a matter of fact, it is of prime importance to have available efficient evaluation algorithms so that the gain in terms of expressiveness be not counterbalanced by a severe performance degradation. So far, fuzzy query processing relies on a technique first introduced by Bosc and Pivert in [2], called *derivation*, whose basic principle is to compute a Boolean query from the fuzzy one to be evaluated. The derived Boolean query, which corresponds to the desired α-cut of the fuzzy query to be evaluated (or a superset thereof), can then be efficiently processed by a regular DBMS, and the satisfaction degree of each answer can be computed either by means of an external program or by a call to user functions in the derived query (which proves rather costful). In the present paper, we introduce an alternative processing strategy, based on the use of *fuzzy views*. Two variants are studied: one where the views are materialized, the other where they are kept abstract.

The remainder of the paper is organized as follows. In Section II, we recall some useful notions about fuzzy relational queries and their evaluation. We also provide a brief survey of related work. Sections III and IV are devoted to the description of the two approaches involving materialized and abstract views respectively. In Section V, we present an

case the join condition involves a fuzzy comparison operator). The principal differences concern mainly two points:

- the filtering of the result, which can be achieved through a number of desired answers (k), a minimal level of satisfaction (α), or both, and
- the nature of the authorized conditions as mentioned previously.

Therefore, the SQLf base block is expressed as:

select [**distinct**] [$k \mid \alpha \mid k, \alpha$] *attributes*
from *relations* **where** *fuzzy-cond*

where *fuzzy-cond* may involve both Boolean and fuzzy predicates. From a conceptual point of view, this expression is interpreted as:

- the fuzzy selection (by *fuzzy-cond*) of the Cartesian product of the relations appearing in the *from* clause,
- a projection over the attributes of the *select* clause (duplicates are kept by default, and if *distinct* is specified the maximal degree among the duplicates is retained),
- the filtering of the result (top k elements and/or those whose score is over the threshold α).

Besides, SQLf also preserves (and extends) the constructs specific to SQL, e.g. nesting operators, relation partitioning, etc., see [14] for more detail. In the following, we only consider single-block Selection-Projection-Join SQLf queries.

Any fuzzy querying system must provide users with a convenient way to define the fuzzy terms that they wish to include in their queries. In practice, the membership function associated with a fuzzy set F is often chosen to be of a trapezoidal shape. Then, F may be expressed by a quadruplet (a, b, c, d) where $core(F) = [b, c]$ and $support(F) = (a, d)$. In a previous work [15], we described a graphical interface aimed at helping the user express his/her fuzzy queries.

B. About Fuzzy Query Processing: Related Work

Fuzzy query processing [8], [12], [9], [3] raises several issues, among which the main two ones are listed hereafter:

- it is not possible to directly use classical indexes for evaluating fuzzy selection conditions;
- an extra step devoted to the computation of the degrees and the filtering of the result is needed, which induces an additional cost.

Then, implementing a fuzzy querying system can be done according to three main types of architectures [17]:

- *loose coupling*: the new features are integrated through a software layer on top of the RDBMS. The main advantage of this type of architecture lies in its portability, which allows connecting to any RDBMS. Its disadvantages include scalability and performance.
- *mild coupling*: the new features can be integrated through stored procedures using either a procedural language for relational databases such as Oracle PL/SQL, or through calls to external functions. With this type of solution, the data are handled only using tools internal to the RDBMS, which entails better performances. Above all, fuzzy queries are directly submitted to the DBMS.
- *tight coupling*: the new features are incorporated into the RDBMS inner core. This solution, which is of course the most efficient in terms of query evaluation, implies to entirely rewrite the query engine (including the parser and query optimizer), which is a quite heavy task.

In [15], we presented an implementation at the junction between mild coupling and tight coupling where i) the membership functions corresponding to the user-specified fuzzy predicates are defined as stored procedures and ii) the gradual connectives (fuzzy conjunction, disjunction, and quantifiers) are implemented in C and integrated in the query processing engine of the RDBMS PostgreSQL.

Since commercial RDBMSs are not able to natively interpret fuzzy queries, some previous works (see [2], [6]) suggested to perform a *derivation step* in order to generate a regular Boolean query used to prefilter the relation (or Cartesian product of relations) concerned. The idea is to restrict the degree computation phase only to those tuples that belong to the α-cut of the result of the fuzzy query (assuming that α is a qualitative threshold specified by the user; 0^+ is used by default). With this type of evaluation strategy, fuzzy query processing involves three steps:

1) derivation of a Boolean query from the fuzzy one, using the threshold α and the membership functions of the fuzzy predicates involved in the *where* clause;
2) processing of the derived query, which produces a classical relation;
3) computation of the satisfaction degree attached to each tuple (followed by a tuple filtering step if necessary[1]), which yields the fuzzy relation that constitutes the final result.

In terms of performances, the interest of using a mild coupling architecture lies in the fact that the resulting fuzzy relation is directly computed during the tuple selection phase (no external program needs to be called to perform step 3).

In the following, we present an original fuzzy query processing strategy relying on the use of *fuzzy views*. The use of fuzzy views has already been advocated in the literature to evaluate fuzzy queries, see [7], [10], [11], [13]. However, in all of these works, the fuzzy views considered correspond either to predicates of a predefined expert vocabulary — whereas we consider *end-user-defined* fuzzy terms — or to previously executed queries. Consequently, these approaches need to perform an *approximate rewriting* of the user query in terms of the views available, and there is no guarantee of completeness/relevance of the set of answers obtained.

---

[1]This filtering step is necessary when the *where* clause of the fuzzy query involves connectives such as means (or trade-off operators in general). Then, the derivation process is said to be *weak*, which means that a *superset* of the actual α-cut may be returned by the derived query (instead of the exact α-cut).

III. An Approach Based on Materialized Views

We first discuss the concept of view in a database context, then we present an approach exploiting *materialized views* in a fuzzy querying context.

A. General Principle

In a database context, a *view* is a *virtual* table derived from the relations present in the database by means of a query. A set of views on a database can thus be seen as an abstraction of the actual schema (considered at the physical level) and provides a *logical schema* representing the links between the data in a more interpretable way.

As such, views cannot be used to precompute query results. A view is built from a query, but this query is executed only when needed. It is kept in memory, and the name of the view constitutes an alias of the query that will be used when this view is called by another query, through a rewriting mechanism.

On the other hand, *materialized views* are views whose underlying query *is* executed, and whose results are stored in a table. Materialized views are thus treated as regular tables by the system, and one may access their data in the same way as one does for base tables. The query that was used to generate the view is still saved in memory, which makes it possible to update the view according to the changes that have been made to the underlying relations.

In the context considered here, i.e., that of fuzzy querying, the first idea we advocate is to use (fuzzy) materialized views for optimizing query processing. The general idea is to use views to store the satisfaction degrees associated with the tuples, instead of using functions to calculate them — which is problematic for query optimization, due to some restrictions in the optimizers of the current commercial DBMSs: the use of functions prevents the DBMS to use indexes and forces the DBMS to scan the entire tables. In order to illustrate our proposals, we will use a database describing flights and airports, involving the relations *Flights*(*Fid*, *depDate*, *depTime*, *arrTime*, *depA*, *arrA*) and *Airport*(*Aid*, *attendance*, *city*, *area*). An example of content is given in Table I.

TABLE I
RELATIONS FLIGHTS AND AIRPORTS FROM DATABASE $\mathcal{DB}$

Flights					
Fid	depDate	depTime	arrTime	depA	arrA
1	02-18-2011	12:15	16:32	CDG	BEY
2	03-22-2012	14:17	22:22	JFK	CDG
3	05-11-2014	4:12	10:14	TLS	YUL

Airports			
Aid	attendance	city	area
CDG	2500	Roissy	13 000
JFK	7000	New York	16 000
TLS	1800	Toulouse	7500
YUL	2200	Montreal	25 000
BEY	1900	Beyrouth	12 000

Let $\mathcal{DB}$ be a database made of the relations $R_1, \ldots, R_n$. We denote by $R_i(A_{i_1}, \ldots, A_{i_{r_i}})$ a relation, $att(R_i) =$
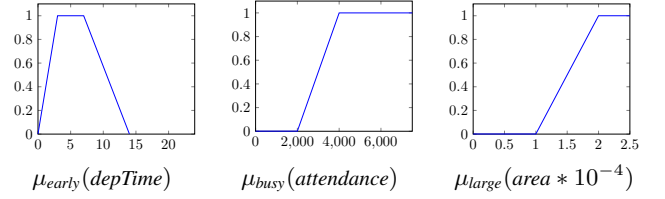

Fig. 1. Membership functions of the terms *early*, *busy*, and *large*

TABLE II
VIEWS ASSOCIATED WITH $\mathcal{DB}$ (MODEL 3)

V_{early}	
Fid	mu
1	0.2
2	0
3	1

V_{busy}	
Aid	mu
CDG	0.5
JFK	1
TLS	0
YUL	0.1
BEY	0

V_{large}	
Aid	mu
CDG	0.3
JFK	0.6
TLS	0
YUL	1
BEY	0.2

$\{A_{i_1}, \ldots, A_{i_{r_i}}\}$ the set of its attributes, and $key(R_i) \subseteq att(R_i)$ the subset of attributes constituting its primary key. We denote by I_R the set of tuples of R. For a tuple $t \in R$, $t[\mathcal{A}]$ where $\mathcal{A} \subseteq att(R)$, is t limited to its values on $\mathcal{A}$.

A fuzzy predicate P_k is represented by a triple (R_i, A_j, μ_k) where R_i is the relation concerned by the predicate, $A_j \in att(R_i)$ is the attribute of R_i to which the predicate applies, and μ_k is the membership function associated with P_k. In the following, we denote these components respectively by $P_k.R$, $P_k.A$, and $P_k.\mu$. Let $\mathcal{F} = \{P_1, \ldots, P_m\}$ be the set of fuzzy terms defined by the current user. We denote by $\mathcal{F}(R_i) = \{(R_i, A, \mu) \in \mathcal{F}\}$ the set of fuzzy terms applying to an attribute from relation R_i. We have of course $\forall i \in [1..n], \mathcal{F}(R_i) \subseteq \mathcal{F}$.

The database $\mathcal{DB}_{pers}$ obtained when converting $\mathcal{DB}$ into a model involving materialized degrees is defined by:

$$\mathcal{DB}_{pers} = (R_1, \ldots, R_n, V_{P_1}, \ldots, V_{P_m})$$

where $\{P_1, \ldots, P_m\} = \mathcal{F}$ and each view V_{P_i} contains the satisfaction degrees related to the fuzzy term P_i. Each view V_{P_i} is built as follows:

- $key(V_{P_i}) = key(R)$ where R is the table to which P_i applies;
- $att(V_{P_i}) = key(R) \cup \{mu\}$.

Hence, the schema of V_{P_i} is the set $key(R) \cup mu$ and its extension is:

$$I_{V_{P_i}} = \{\langle t[key(R)], P_i.\mu(t[P_i.A])\rangle \mid t \in R\}.$$

Example 1. Let us consider the database introduced above and the fuzzy terms *early*, *large* and *busy* (cf. Figure 1) associated respectively with the attributes *Flights.depTime*, *Airports.area*, and *Airports.attendance*. We get:

$$\mathcal{DB}_{pers} = (\textit{Flights}, \textit{Airports}, V_{early}, V_{busy}, V_{large}).$$

The three views are represented in Table II.

B. Query Processing Strategy

We now deal with fuzzy query processing in the context of a database built according to the principles suggested above. The fuzzy query language we consider is SQLf [5], [14] limited to single block projection-selection-join queries. More precisely, the fuzzy queries considered are of the form:

select $[\alpha;]$ $\mathcal{A}$ **from** $\mathcal{R}$ **where** C

where α is a threshold corresponding to the minimal satisfaction degree authorized by the user (by default, $\alpha = 0^+$), $\mathcal{A}$ is a set of attributes, $\mathcal{R}$ is a set of joined relations, and C is a fuzzy condition that may take one of the following forms:

- $A_1 \theta A_2$, where A_1 and A_2 are attributes, and θ is a fuzzy comparison operator;
- $A \theta val$, where A is an attribute, $val \in dom(A)$, and θ is a fuzzy comparison operator;
- A **is** P, where A is an attribute and P is a fuzzy predicate;
- $agg(C_1, \ldots, C_p)$ where every C_i is a fuzzy condition and agg is a fuzzy connective.

Let $\mathcal{Q}_{\mathcal{DB}}$ be a query of the form defined above, addressed to the database $\mathcal{DB}$. The idea is to evaluate $\mathcal{Q}_{\mathcal{DB}}$ by means of a query $deriv(\mathcal{Q}_{\mathcal{DB}}, \alpha)$, derived from $\mathcal{Q}_{\mathcal{DB}}$ and sent to $\mathcal{DB}_{pers}$. In the following, we use two example queries denoted by Q_1 and Q_2 (cf. listings 1 and 2). Query Q_1 is a simple selection-projection query while Q_2 also includes a join. In Q_1, α is not specified, thus its value is 0^+, whereas in Q_2, $\alpha = 0.5$ is specified.

```
1  SELECT Fid, depDate, depTime, arrTime
2  FROM Flights WHERE depTime IS early
```

Listing 1. Query Q_1

Query Q_1 returns the flights whose departure time is considered "early" by the user, according to his/her definition of the fuzzy term *early*.

```
1  SELECT 0.5; Fid, depDate, depTime, arrTime, name, city
2  FROM Flights JOIN Airports
3    ON (Flights.depA=Airports.name)
4  WHERE depTime IS early AND area IS large
```

Listing 2. Query Q_2

Query Q_2 returns the flights whose departure time is considered "early" and that leave from an airport whose area is considered "large" by the user, with a global satisfaction degree at least equal to 0.5.

The query $deriv(Q_{DB}, \mathcal{DB}_{pers})$ has the following form:

select $[\alpha;]$ $\mathcal{A}$, *mu* **from** $compl(\mathcal{R})$ **where** $deriv(C, \alpha)$

where *mu* is the expression used to compute the final degree attached to an answer and $compl(\mathcal{R})$ is defined as follows:

$$compl(\mathcal{R}) = \{R \bowtie_{key(R)} \Gamma \mid R \in \mathcal{R} \wedge \\ \Gamma \in \{V_{P_i} \mid P_i \in \mathcal{F}(Q) \cap \mathcal{F}(R)\}$$

For instance, query Q_1 is rewritten into:

```
1  SELECT Fid, depDate, depTime, arrTime, V_early.mu
2  FROM Flights NATURAL JOIN V_early
3  WHERE V_early.mu > 0
4  ORDER BY V_early.mu DESC
```

Listing 3. Rewriting of query Q_1 (FMV approach)

As to query Q_2, it is rewritten into:

```
1  SELECT Fid, depDate, depTime, arrTime, Aid, city,
2  least(V_early.mu, V_large.mu) AS degree
3  FROM Flights JOIN Airports
4  ON (Flights.depA = Airports.Aid)
5  NATURAL JOIN V_early NATURAL JOIN V_large
6  WHERE V_early.mu > 0.5 AND V_large.mu > 0.5
7  ORDER BY degree DESC
```

Listing 4. Rewriting of query Q_2 (FMV approach)

IV. AN APPROACH BASED ON ABSTRACT VIEWS

In this second approach, the views are not materialized anymore, they just correspond to *named queries* memorized in the DBMS. For instance, the view *F_early* associated with the fuzzy term *early* is defined as

```
1  create view V_early AS
2  SELECT Fid, depDate, depTime, arrTime,
3      depA, arrA, f_early(depTime) AS mu
4  FROM Flights
5  WHERE depTime between 0 AND 14.
```

Listing 5. Rewriting of query Q_1 (FAV approach)

where *f_early* is a user function that encodes the membership function of the fuzzy predicate *early*. In this view definition, the *where* clause is used to restrict the search to those tuples that somewhat satisfies the predicate (i.e., that have a *depTime* value in the support of the fuzzy term *early*).

The query $deriv(Q_{DB}, \mathcal{DB}_{pers})$ is now as follows:

select $[\alpha;]$ $\mathcal{A}$, *mu* **from** $\mathcal{R}'$ **where** $deriv(C, \alpha)$

where *mu* is the expression used to compute the final degree attached to an answer and $\mathcal{R}'$ is defined as:

$$\mathcal{R}' = \{R_i \in \mathcal{R} \mid \mathcal{F}(Q) \cap \mathcal{F}(R_i) = \emptyset\} \cup \\ \{V_{R_i} \mid R_i \in \mathcal{R} \wedge \mathcal{F}(Q) \cap \mathcal{F}(R_i) \neq \emptyset\}$$

For instance, query Q_1 is rewritten into:

```
1  SELECT Fid, depDate, depTime, arrTime,
2  f_early(depTime) AS mu
3  FROM V_early
4  ORDER BY mu DESC
```

Listing 6. Rewriting of query Q_1 (FAV approach)

Here, $deriv(C, \alpha)$ corresponds to *true* since the initial fuzzy query does not involve any user-defined threshold (α is equal to the default value 0^+).

As to query Q_2, it is rewritten into:

```
1  SELECT Fid, depDate, depTime, arrTime, Aid, city,
2  least(early(depTime), large(area)) AS mu
3  FROM V_early JOIN V_large
4  ON (V_early.depA = V_large.Aid)
5  WHERE (V_early.depTime between 2 AND 11)
6      AND (V_large.area > 1.5)
7  ORDER BY mu DESC
```

Listing 7. Rewriting of query Q_2 (FAV approach)

Here, [2, 11] (resp. $[1.5, +\infty)$) corresponds to the 0.5-cut of the fuzzy term *early* (resp. *large*).

Remark 1. In order to avoid unnecessary joins between views defined on the same base table, a solution consists in creating the views at runtime. Then a fuzzy view can be associated with a conjunction of fuzzy criteria on the same table.

V. EXPERIMENTAL COMPARISON

In this section, we discuss the results of some experimentations that we carried out, aimed to compare the two approaches with each other on the one hand, and with a classical function-based derivation strategy on the other hand. Two criteria will be used to compare the solutions: i) the query processing time; ii) the size of the personalized database $\mathcal{DB}_{pers}$.

In order to compare the approaches, we used excerpts of a real-world database describing the domestic flights in the USA between 1987 and 1989 [2]. This database was managed using the RDBMS PostgreSQL 9.4 [3].

In order to assess the performances of the different strategies, we built a set of eight queries. The first six have different levels of complexity — depending on the number of fuzzy terms (from one to five) and relations (one or two) involved — but they all use a threshold equal to 0^+. The last two use a threshold value equal to 0.5. As an example, we give the most complex one hereafter, which returns the long distance flights that leave early, arrive early, and whose departure airport is located in the North-East of the US:

```
1  SELECT 0.5; id_flight
2  FROM Flights JOIN Airports
3    ON Flights.origin = Airports.iata
4  WHERE distance IS long
5  AND depTime IS early_dep AND arrTime IS early_arr
6  AND latitude IS north AND longitude IS east
```

Listing 8. Query 6: two relations, five fuzzy terms

In the function-based (mild coupling) approach described in [15] and implemented in the PostgreSQLf prototype, this query would be expressed as follows:

```
1  SELECT 0.5; id_flight, get_mu() AS degree
2  FROM Flights JOIN Airports
3    ON Flights.origin = Airports.iata
4  WHERE distance ~= long && depTime ~= early_dep
5    && arrTime ~= early_arr
6  ORDER BY degree DESC
```

Listing 9. Query 6 in the function-based approach

Here, *long*, *early_dep* and *early_arr* are user-functions called in the *where* clause, which prevents the DBMS from using indexes, thus makes the query costly to execute (sequential scans are performed).

Four $\mathcal{DB}$ sizes were considered, as well as three vocabularies. Their characteristics are given in Table III. The results obtained for a sample of four queries are represented in Table IV. The main conclusions are the following:

- as expected, there is an important increase in the size of the database when materialized views are used. The ratio $\frac{|\mathcal{DB}_{pers}|}{|\mathcal{DB}|}$ varies between 4.25 (for the smallest vocabulary) and 7.12 (for the largest vocabulary). Of course, no such overhead exists with abstract views.
- creating/updating the materialized views takes a significant amount of time (between 16 and 40 seconds) whereas creating abstract views is immediate.
- in terms of query processing time, there is no clear winner between these two approaches. The somewhat

[2]http://stat-computing.org/dataexpo/2009/
[3]We used a laptop with a 3.1 GHz Intel Core i7 and 16GB of RAM.

TABLE III
DATABASES (TOP) AND VOCABULARIES (BOTTOM) USED

Database	Nb of tuples	Size in bytes
flights_150k	150,000	20,013,056
flights_200k	200,000	27,254,784
flights_250k	250,000	34,111,488
flights_300k	300,000	40,853,504

Vocabulary	Nb of atributes	Nb of terms
voc_flights_sm	8	35
voc_flights_med	12	51
voc_flights_lar	17	73

disappointing behavior of the materialized-view-based approach comes from the fact that it is necessary to perform a join between a materialized view and the corresponding base table for each fuzzy condition in the query.

Table V shows the processing times obtained with a mild coupling strategy (using the prototype PostgreSQLf described in [15]). As can be seen, it is clearly outperformed by both view-based approaches.

Finally, it appears that using materialized views is not an interesting solution as they induce an important increase in terms of storage space and do not yield significantly better query processing times than abstract views. Let us not forget either that materialized views are problematic from a DB update perspective, since every modification of the data in a base table makes it necessary to recalculate the associated fuzzy views. On the other hand, the approach based on abstract views appears to be a promising solution: it is more efficient than a mild coupling approach such as PostgreSQLf [15] and does not induce any extra cost in terms of data storage. It is all the more interesting as it is completely portable: its use does not require to modify the underlying DBMS, but only implies to add a simple software layer on top of it.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we have defined a new approach to the processing of fuzzy relational queries (of the type projection-selection-join), based on the used of *fuzzy views*. These views can be materialized or not. In the first case, the idea is to store in the database itself the degrees reflecting the satisfaction of the user-defined fuzzy predicates by the tuples from the tables concerned. In the second case, the views are kept abstract and are used to efficiently access the tuples that belong to the α-cut of the query result, by means of a derived Boolean selection condition.

The experiments that we have performed on a real-world database have shown the pros and cons of each approach. It appears that materialized views induce an important increase in storage space, which makes it rather unrealistic to use in practice. On the other end, the solution based on abstract views — which does not have this drawback — yields performances significantly better than the state-of-the-art approach based on a mild coupling strategy, and has the advantage of being completely portable.

TABLE IV
Experimental results: Materialized Views (top) and Abstract Views (bottom)

Vocabulary	Database	Size of $\mathcal{DB}_{pers}$	Time to build $\mathcal{DB}_{pers}$	Time Q_1	Time Q_2	Time Q_3	Time Q_4
voc_flights_sm	flights_150k	87,212,032	16.3 s	128.5 ms	117.7 ms	68.6 ms	87.6 ms
	flights_200k	117,342,208	19.5 s	136.6 ms	133.6 ms	108.5 ms	128.5 ms
	flights_250k	144,941,056	24.3 s.2	199.4 ms	219.7 ms	149.0 ms	151.4 ms
	flights_300k	177,094,656	30.7 s	234.5 ms	236.1 ms	174.6 ms	205.3 ms
voc_flights_med	flights_150k	113909760	17.6 s	126.9 ms	124.4 ms	74.4 ms	93.8 ms
	flights_200k	153,190,400	23.9 s	152.4 ms	128.9 ms	106.8 ms	137.5 ms
	flights_250k	189,407,232	28.1 s	198.3 ms	212.1 ms	148.1 ms	155.0 ms
	flights_300k	232,759,296	35.3 s	234.9 ms	236.3 ms	175.3 ms	211.0 ms
voc_flights_lar	flights_150k	139,689,984	19.9 s	128.8 ms	123.0 ms	65.5 ms	85.9 ms
	flights_200k	193,290,240	25.9s	193.8 ms	181.8 ms	108.4 ms	138.1 ms
	flights_250k	240,271,360	32.9 s	237.4 ms	231.5 ms	144.3 ms	154.5 ms
	flights_300k	291,176,448	40.4 s	260.7 ms	258.0 ms	171.3 ms	209.3 ms

Database	Time Q_1	Time Q_2	Time Q_3	Time Q_4
flights_150k	118.0 ms	169.7 ms	138.5 ms	50.3 ms
flights_200k	199.0 ms	219.0 ms	191.3 ms	76.6 ms
flights_250k	238.7 ms	229.5 ms	185.8 ms	80.5 ms
flights_300k	257.5 ms	375.0 ms	276.7 ms	123.8 ms

TABLE V
Experimental results with a mild coupling strategy

Database	Time Q_1	Time Q_2	Time Q_3	Time Q_4
flights_150k	569.6ms	512.9 ms	519.6 ms	536.5 ms
flights_200k	729.3 ms	705.2 ms	716.5 ms	723.3 ms
flights_250k	781.2 ms	913.5 ms	910.2 ms	912.3 ms
flights_300k	946.4 ms	1103.8 ms	1109.5 ms	1147.3 ms

A perspective is to study the way materialized views could be exploited, along with some statistics maintained by commercial DBMSs, in order to efficiently process fuzzy queries involving a *quantitative threshold* (instead of a qualitative one as considered here), in the spirit of top-k queries.

References

[1] P. Bosc, B. Buckles, F. Petry, and O. Pivert. Fuzzy databases. In J. Bezdek, D. Dubois, and H. Prade, editors, *Fuzzy Sets in Approximate Reasoning and Information Systems, The Handbook of Fuzzy Sets Series*, pages 403–468. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[2] P. Bosc and O. Pivert. Some algorithms for evaluating fuzzy relational queries. In B. Bouchon-Meunier and R. Yager, editors, *Lecture Notes in Computer Science 521 (Proc. of IPMU'90)*, pages 431–442. Springer-Verlag, 1991.

[3] P. Bosc and O. Pivert. On the evaluation of simple fuzzy relational queries: principles and measures. In R. Lowen and M. Roubens, editors, *Fuzzy logic – State of the Art*, pages 355–364. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.

[4] P. Bosc and O. Pivert. On the efficiency of the alpha-cut distribution method to evaluate simple fuzzy relational queries. In B. Bouchon-Meunier, R. Yager, and L. Zadeh, editors, *Advances in Fuzzy Systems — Applications and Theory Vol. 4: Fuzzy Logic and Soft Computing*, pages 251–260. World Scientific Publishing, Singapore, 1995.

[5] P. Bosc and O. Pivert. SQLf: a relational database language for fuzzy querying. *IEEE Trans. on Fuzzy Systems*, 3:1–17, 1995.

[6] P. Bosc and O. Pivert. SQLf query functionality on top of a regular relational database management system. In O. Pons, M. Vila, and J. Kacprzyk, editors, *Knowledge Management in Fuzzy Databases*, pages 171–190. Physica-Verlag, Heidelberg, Germany, 2000.

[7] P. Buche and S. Loiseau. Using contextual fuzzy views to query imprecise data. In T. J. M. Bench-Capon, G. Soda, and A. M. Tjoa, editors, *Database and Expert Systems Applications, 10th International Conference, DEXA '99, Florence, Italy, August 30 - September 3, 1999, Proceedings*, volume 1677 of *Lecture Notes in Computer Science*, pages 460–472. Springer, 1999.

[8] J. Galindo, J. M. Medina, O. Pons, and J. C. Cubero. A server for fuzzy SQL queries. In *Proc. of FQAS'98*, pages 164–174, 1998.

[9] M. Goncalves and L. Tineo. SQLf3: An extension of SQLf with SQL features. In *Proc. of FUZZ-IEEE'01*, pages 477–480, 2001.

[10] A. HadjAli and O. Pivert. Towards fuzzy query answering using fuzzy views - A graded-subsumption-based approach. In A. An, S. Matwin, Z. W. Ras, and D. Slezak, editors, *Foundations of Intelligent Systems, 17th International Symposium, ISMIS 2008, Toronto, Canada, May 20-23, 2008, Proceedings*, volume 4994 of *Lecture Notes in Computer Science*, pages 268–277. Springer, 2008.

[11] H. Jaudoin and O. Pivert. Rewriting fuzzy queries using imprecise views. In J. Eder, M. Bieliková, and A. M. Tjoa, editors, *Advances in Databases and Information Systems - 15th International Conference, ADBIS 2011, Vienna, Austria, September 20-23, 2011. Proceedings*, volume 6909 of *Lecture Notes in Computer Science*, pages 257–270. Springer, 2011.

[12] J. Kacprzyk and S. Zadrożny. FQUERY for ACCESS: fuzzy querying for a Windows-based DBMS. In P. Bosc and J. Kacprzyk, editors, *Fuzziness in Database Management Systems*, pages 415–433. Physica Verlag, 1995.

[13] W. Labbadi and J. Akaichi. Answering conjunctive fuzzy query using views: efficient algorithm to return the best k-tuples pervasive healthcare application. *IJMEI*, 7(4):293–312, 2015.

[14] O. Pivert and P. Bosc. *Fuzzy Preference Queries to Relational Databases*. Imperial College Press, London, UK, 2012.

[15] G. Smits, O. Pivert, and T. Girault. Towards reconciling expressivity, efficiency and user-friendliness in database flexible querying. In *Proc. of the IEEE International Conference on Fuzzy Systems (FUZZ-IEEE'13)*, 2013.

[16] V. Tahani. A conceptual framework for fuzzy query processing — a step toward very intelligent database systems. *Information Processing and Management*, 13(5):289–303, 1977.

[17] A. Urrutia, L. Tineo, and C. Gonzalez. FSQL and SQLf: Towards a standard in fuzzy databases. In J. Galindo, editor, *Handbook of Research on Fuzzy Information Processing in Databases*, pages 270–298. Information Science Reference, Hershey, PA, USA, 2008.