Increasing the Performance of Fuzzy Retrieval Using Impact Ordering

Carlos D. Barranco[1] Sven Helmer[2]

[1]Division of Computer Science, School of Engineering, Pablo de Olavide University
Seville, Spain
[2]School of Computer Science and Information Systems, Birkbeck, University of London
London, United Kingdom
Email: cbarranco@upo.es, sven@dcs.bbk.ac.uk

Abstract— *We propose an approach for indexing fuzzy data based on inverted files that speeds up retrieval considerably by stopping the traversal of postings lists early. This is possible because the entries in the postings lists are organized in a way that guarantees that there are no matching items beyond a certain point in a list. Consequently, we can reduce the number of false positives significantly, leading to an increase in retrieval performance. We have implemented our approach and evaluated it experimentally, comparing it to an approach that has previously been shown to be superior to other methods.*

Keywords— fuzzy databases, access methods, inverted files, physical design

1 Introduction

Being able to handle imprecise or uncertain data becomes ever more important in today's world. There are numerous applications that have to manage imperfect data in areas such as knowledge discovery, mediator systems, information retrieval, multimedia, and profiling (vague) users' preferences. One way to model imprecision and uncertainty is to employ concepts from fuzzy set theory. Fuzzy sets are making their way into the database world, evidenced by the fact that several proposals for fuzzy database models and systems exist [1, 2, 3, 4, 5] (for an overview see [6]).

We are focusing on a well-known model for fuzzy databases, namely the possibilistic database model [5]. This approach uses the concepts of possibility and necessity measures for flexible querying. While this is an elegant way to formulate queries, indexing possibility distributions is not straightforward. Bosc and Galibourg introduced an indexing principle based on α-cuts (i.e. the elements of a fuzzy set with a membership degree of at least α) [7]. The parameter α is tied directly to a threshold value determined by a user defining a query, which makes it unrealistic to materialize dozens or even hundreds of α-cuts in different indexes to be able to answer various queries efficiently. For that reason Bosc and Galibourg developed a filtering mechanism based on the supports and cores of the indexed possibility distributions[1]. However, this can introduce a large number of false positives, as the filter only acts as a "quick-and-dirty" test. All the candidates that pass the filter have to be checked for eligibility in a second step.

We propose an access method for indexing possibility distributions that cuts down the number of false positives significantly. Many different α-cuts are stored compactly in a single index structure by arranging the references to possibility distributions in a clever way. The idea is based on impact ordering of inverted files used in information retrieval (IR). In IR systems, document references are sorted in descending order according to the term frequency value to make sure that important documents for a query are processed first [8]. In our case references to possibility distributions are sorted according to the certainty degrees of their elements, making it possible to stop scanning lists of references at an early stage (because we know that no eligible data items can follow). By applying customized gap compression schemes to the inverted files, we can store the lists very compactly.

The remainder of this paper is organized as follows. In the section immediately following this introduction, we cover basic definitions of possibilistic database systems and how to query them. In Section 3 we describe our access method in more detail and explain how to answer queries based on the possibility measure and those based on the necessity measure. Section 4 includes a brief description of the approach that we compared ours to. This is followed by a section on the experimental evaluation, specifying the environment in which this evaluation was run and presenting the results of the evaluation. We wrap up the paper with a conclusion and outlook in Section 6.

2 Preliminaries

Before going into details about indexing, let us briefly define possibility distributions and their application in flexible querying as defined in [5].

2.1 Possibilistic Databases

We assume (without loss of generality) n data items $(o_1, o_2, \ldots, o_n)$ in our database, all of which have an attribute A with a discrete domain Ω. The value of attribute A of data item o_i is described by a possibility distribution function $\pi_{A(o_i)}$ on Ω.

The possibility distribution $\pi_{A(o_i)}$ is a set of possible values for the attribute A of the data item o_i (together with the certainty of each value) and is defined as:

$$\pi_{A(o_i)} : \Omega \mapsto [0, 1] \tag{1}$$

$\pi_{A(o_i)}(\omega_j) = 0$ (for $\omega_j \in \Omega$) means that it is *impossible* that the attribute A can take on the value ω_j. $\pi_{A(o_i)}(\omega_j) = 1$, on the other hand, means that it is *completely* possible that A can take on the value ω_j. If we want to express that it might

[1]Supports and cores are special α-cuts.

ISBN: 978-989-95079-6-8

ω_1	$\rightarrow$	$\alpha_{1,1}$	$r(o_1^{1,1}), r(o_2^{1,1}), r(o_3^{1,1}), \ldots$	$\alpha_{1,2}$	$r(o_1^{1,2}), r(o_2^{1,2}), r(o_3^{1,2}), \ldots$	$\ldots$	$\ldots$
ω_2	$\rightarrow$	$\alpha_{2,1}$	$r(o_1^{2,1}), r(o_2^{2,1}), r(o_3^{2,1}), \ldots$	$\alpha_{2,2}$	$r(o_1^{2,2}), r(o_2^{2,2}), r(o_3^{2,2}), \ldots$	$\ldots$	$\ldots$
$\vdots$							
ω_m	$\rightarrow$	$\alpha_{m,1}$	$r(o_1^{m,1}), r(o_2^{m,1}), r(o_3^{m,1}), \ldots$	$\alpha_{m,2}$	$r(o_1^{m,2}), r(o_2^{m,2}), r(o_3^{m,2}), \ldots$	$\ldots$	$\ldots$

Figure 1: Inverted file index for fuzzy retrieval

be possible for A to take on the value ω_j, then we can assign a value from the interval $(0, 1)$ to $\pi_{A(o_i)}(\omega_j)$, depending on how plausible we think it is that A takes on this value. Not that the possible values for A are mutually exclusive, i.e. A takes on a single value, we are just uncertain about which one is the correct value. For reasons of consistency, $\pi_{A(o_i)}$ should be normalized, i.e. $\exists\omega \in \Omega : \pi_{A(o_i)}(\omega) = 1$.

As already mentioned, we assume that Ω is discrete, since we are interested in indexing scalar data. In the case of applications using continuous numerical data, we either have to discretize the possibility distribution or use a different kind of index structure [9]).

2.2 Flexible Querying

We not only allow imprecision in the data, but also flexible querying, i.e. the query condition c determining the acceptable values of an attribute is described by a normalized fuzzy set $\mu_c : \Omega \mapsto [0, 1]$.

In order to be able to check whether a data item satisfies a query condition, we need to introduce the concept of a fuzzy measure. Let X be an element of the power set of Ω ($X \in \mathcal{P}(\Omega)A$). The *possibility* of X, $\Pi(X)$, is measured by looking at the elements of X:

$$\Pi(X) = \max_{\omega \in X} \pi(\omega) \tag{2}$$

Possibility theory uses two concepts to measure the likeliness of X: the possibility measure, as already described above, $\Pi(X)$ and the necessity measure $N(X)$. The *necessity* of X, $N(X)$, is defined by the unlikeliness of the complement of X:

$$N(X) = 1 - \Pi(\overline{X}) \tag{3}$$

What does this mean for an attribute value satisfying a query condition c? The possibility of doing so is measured with the help of the possibility measure:

$$\Pi(c|A(o_i)) = \max_{\omega \in \Omega} \min(\mu_c(\omega), \pi_{A(o_i)}(\omega)) \tag{4}$$

Informally speaking, $\Pi(c|A(o_i))$ returns the highest degree to which $A(o_i)$ can possibly satisfy c.

The necessity of an attribute value satisfying query condition c is defined as:

$$N(c|A(o_i)) = \min_{\omega \in \Omega} \max(\mu_c(\omega), 1 - \pi_{A(o_i)}(\omega)) \tag{5}$$

$N(c|A(o_i))$, on the other hand, returns the degree to which $A(o_i)$ certainly satisfies c.

Usually users are interested in a small subset of data items in the database. Selective queries can be formulated by providing an acceptance threshold α (e.g. return all data items whose attribute values possibly satisfy c to at least a degree of 0.8: $\{o_i|\Pi(c|A(o_i)) \geq 0.8\}$). Furthermore, using the necessity measure also leads to more selective queries than using the possibility measure.

3 Indexing Possibility Distributions

As mentioned in Section 1 we use an inverted file index to index possibility distributions. An inverted file consists of a *directory* containing all distinct values $\omega_1, \omega_2, \ldots, \omega_m$ of the domain Ω and a *list* (also called a *postings list*) for each value.[2] Within each postings list of the inverted file, we sort the references to data items o_j, denoted by $r(o_j)$, in descending order of the degree of membership of w_i in $\pi_{A(o_j)}$. Since we want to apply gap compression (more on this later in Section 3.3), we arrange the references in blocks, each block preceded by a value $\alpha_{i,k}$, meaning that this is the k-th block of the posting list associated with ω_i. All attributes of the data items referenced in a block satisfy $\pi_{A(o_j)}(\omega_i) \geq \alpha_{i,k}$. Figure 1 illustrates the layout described above; note that the data items have superscripts indicating which block they belong to. The threshold values $\alpha_{i,k}$ divide up the interval [0,1] into equally-sized partitions (i.e. each block covers an equally-sized interval). Within each block all references to data items (in the form of IDs) are sorted in ascending order.

Depending on the query type, we access the inverted file index slightly differently. We will first look at queries based on the possibility measure and then turn to those based on the necessity measure.

3.1 Possibility Measure Queries

If we are searching for all data items o_j whose attribute A satisfies condition c possibly to a degree of at least α, we only have to look up the values ω_i in the inverted file for which $\mu_c(\omega_i) \geq \alpha$ (see Equation (4)). In addition to that, only data items for which $\pi_{A(o_j)}(\omega_i) \geq \alpha$ will qualify. During a search, we scan a postings list until we reach the first block whose value $\alpha_{i,l}$ is smaller than the query threshold value α.[3] If α is between $\alpha_{i,l-1}$ and $\alpha_{i,l}$, then we still have to check the data items referenced in this block whether they are false positives or not (as some of them could satisfy the query predicate). However, only the data items in this last block can be false

[2] For an overview of traditional inverted files see [10].

[3] Obviously, we also stop when we reach the end of a postings list.

ISBN: 978-989-95079-6-8

positives and have to be checked, all other data items referenced in earlier blocks will satisfy the query predicate and can be retrieved without checking for false positives. If α is equal to $\alpha_{i,l-1}$ we do not have to check the items of block $\alpha_{i,l}$. Actually, in this case we do not have to check for false positives at all.

3.2 Necessity Measure Queries

Processing queries based on the necessity measure can be done in two different ways, depending on the cardinality of the domain Ω. For large domains we can simply use possibility measure queries as a filter (we will call this method 'simplified' in the following); for small domains we can use the index described above to determine the answer set (without accessing the data items themselves).

The inequality $N(X) \leq \Pi(X)$ always holds for necessity and possibility measures, i.e. $N(c|A(o_i)) \geq \alpha \Rightarrow \Pi(c|A(o_i)) \geq \alpha$. So instead of directly searching for the data items that satisfy the query condition c necessarily to the degree α, we search for data items satisfying c possibly. We retrieve these data items and check if they also satisfy c necessarily. A drawback of this technique is that we introduce false positives, due to the data items that satisfy c possibly but not necessarily.

Accessing all the candidates returned by the possibility measure query in the simplified processing will cause a lot of random I/O. However, we can use the inverted file index to help us in sorting out false positives returned by a possibility measure search. If we can find an ω_i for a data item o_j such that $\max(\mu_c(\omega_i), 1 - \pi_{A(o_j)}(\omega_i)) < \alpha$, then we know that $A(o_j)$ cannot satisfy c necessarily (more on this in just a moment, see also Equation 5). Having a value for $\mu_c(\omega_i)$ that is greater or equal to α will never result in $\max(\mu_c(\omega_i), 1-\pi_{A(o_j)}(\omega_i)) < \alpha$. So in order to sort out false positives, we have to access the postings lists of the values ω_i for which $\mu_c(\omega_i) < \alpha$. Unfortunately, this means we have to look up the complement of the values ω_i used for the possibility measure query and scan the corresponding postings lists.[4] (Consequently, this will only be efficient for relatively small domains.) During query processing, we keep scanning such a postings list in a blockwise fashion as long as $\alpha_{i,l} > 1 - \alpha$ holds and we stop as soon as this condition is not satisfied anymore.[5] Any references to data items found during this scan can be safely discarded from the candidates determined by the possibility measure query. The reasoning is the following: while $\alpha_{i,l} > 1 - \alpha$, we know that for all data items referenced in this block $\pi_{A(o_j)}(\omega_i) > 1 - \alpha$ ($\Leftrightarrow 1 - \pi_{A(o_j)}(\omega_i) < \alpha$) holds. Together with $\mu_c(\omega_i) < \alpha$, this means that $A(o_j)$ cannot satisfy c necessarily. Once $\alpha_{i,l}$ drops below $1 - \alpha$, we do not have enough information to determine whether the data items referenced in this block (and the following blocks) satisfy c necessarily or not and we stop scanning the list further. All the data items remaining in the candidate set after scanning the lists of the ω_i for which $\mu_c(\omega_i) < \alpha$ have to be fetched and checked for false positives.

[4] This is why we will also call this approach complement-based.

[5] Again, we also stop when reaching the end of a list.

3.3 Index Compression

The reason for arranging the references to data items in blocks is that we can use gap encoding on the ascending sequences of data item IDs. For example, instead of storing the sequence 103,110,114,116,121 we store 103,7,4,2,5. Compared to absolute ID numbers, relative gaps can be stored using less storage space. However, from time to time even gaps can become quite large, i.e. a large gap could need almost as much storage space as a regular ID. So that we do not have to allocate the same space for each gap, we use variable length encoding [11]. The main idea is to use few bits for small gaps and many bits for large gaps.

In order to keep the codewords byte-aligned for faster processing, we use Variable-Byte (VB) code [12]. Each byte contains a so-called continuation bit, which signals if the end of the current codeword has been reached or if we have to continue decoding. Usually the high bit is dedicated to this purpose, while the lower 7 bits of each byte encode gap information. If a gap fits into 7 bits, then we encode this gap in the lower 7 bits and set the continuation bit to 1. Otherwise, we encode the highest bits of the gap and set the continuation bit to 0. The remaining bits are encoded in the same manner. VB-encoding is a good compromise between compression ratio and processing speed. For example, encoding 5 in VB-code would result in **1**0000101, while encoding 824 would result in **0**0000110 **1**0111000 (where the bits in boldface are the continuation bits).

Instead of using floating-point numbers for the threshold levels $\alpha_{i,k}$ directly in the index, we store IDs of threshold levels using a one-byte integer, looking up the floating-point values in a table mapping the IDs to these values. Due to the fixed threshold levels we use throughout the whole index, we can do this. As we will see later in the evaluation section, 256 different threshold levels are more than enough. If more than 256 threshold levels should be needed, we can apply VB-encoding to the threshold levels as well.

4 Comparison to Existing Approach

We use the approach by Bosc and Galibourg [7] as a reference for comparison. There are two special α-cuts, the *core* $L_1(\mu_F)$ and the *support* $L_{>0}(\mu_F)$ of a fuzzy set F. Bosc and Galibourg have shown that the following implications hold (for all $\alpha > 0$):

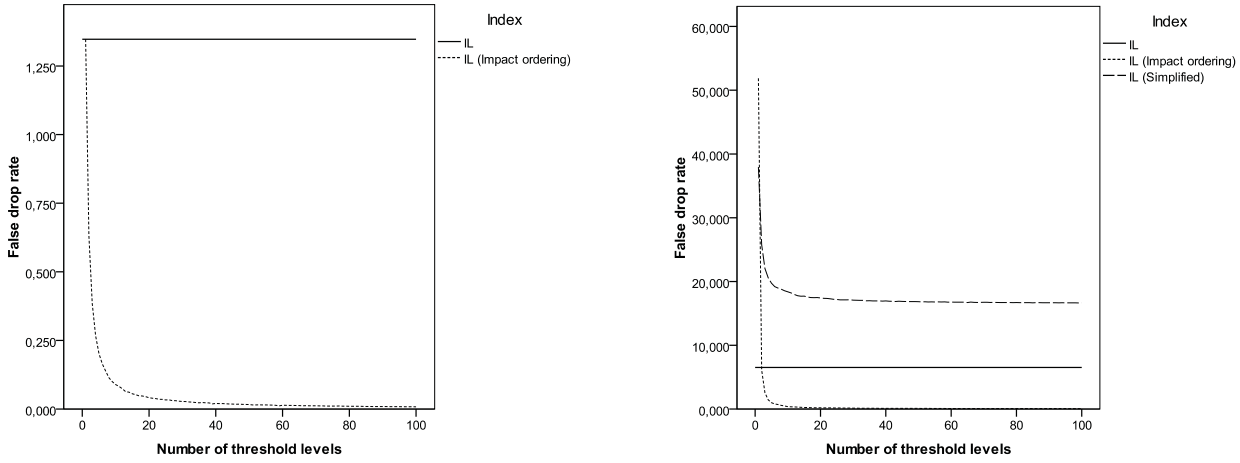
$$\Pi(c|A(o_i)) \geq \alpha \quad \Rightarrow \quad L_{>0}(\pi_{A(o_i)}) \cap L_\alpha(\mu_c) \neq \emptyset \qquad (6)$$

$$N(c|A(o_i)) \geq \alpha \quad \Rightarrow \quad L_1(\pi_{A(o_i)}) \subseteq L_\alpha(\mu_c) \qquad (7)$$

Given a query condition c we have to determine whether the α-cut of μ_c ($L_\alpha(\mu_c)$ intersects with the support of a data item's attribute value (for possibility queries) or if the core of the attribute value is a subset of $L_\alpha(\mu_c)$ (for necessity queries). If this is not the case, then we can safely discard a data item, as it will not satisfy the original query condition.

We implemented this technique using the same framework as for our approach, i.e. using compressed inverted files for indexing the cores and supports of possibility distributions. The index containing the cores stores the cardinality of each core together with the reference to the possibility distribution. In

ISBN: 978-989-95079-6-8



(a) Possibility measure

(b) Necessity measure

Figure 2: False drop rates

this way, superset queries (as described in Equation (7)) can be answered using an inverted file by filtering out all references whose cores have a higher cardinality than the query set. For a more detailed description on how to optimize this kind of data structure, see [13] (comparisons to other data structures can also be found in this paper; the inverted file approach, however, was the data structure with the best performance among all those that were tested).

5 Experimental Evaluation

5.1 Benchmark Environment

The benchmarks were run on a Pentium 4 (3.2 GHz) PC with 1GByte main memory running Windows XP. The index structures and a simulator for the storage system were implemented in Java 1.6. We decided to implement the simulator in Java, as this makes us platform-independent. The simulator allows us to avoid unwanted effects due to caching. We set the block size of the simulator to a page size of 4K, which is the page size of the underlying operating system.

We generated possibility distributions for the data items using the following parameters: the domain size ($|\Omega|$) is equal to 25 and the database size is equal to 100K data items (each data item is associated with a possibility distribution). The elements of a particular possibility distribution are taken randomly from the domain, using a uniform distribution. This means that we first determine the cardinality of the support of a possibility distribution, which is uniformly distributed in [1,22] (22 elements cover 90% of the domain). All the elements in the domain have the same probability of making it into the support. The membership degrees of the chosen elements are uniformly distributed in (0,1.0]. If a possibility distribution is not normalized, the highest degree is converted to 1.0.

In this paper we wanted to focus on typical fuzzy scalar domains, therefore we decided to keep the domain cardinality rather small. This is motivated by the fact that the elements of fuzzy domains usually represent categories. Normally these categories are determined by experts based on the criteria (1) that they should be easy to use by ordinary users and (2) that fuzzy relations (allowing flexible comparisons) can be defined on them. It is much easier to satisfy both criteria with a small domain cardinality; that is why typical fuzzy scalar domains are rather small.

Another important parameter is the clustering of the data items on disk pages. For the results shown in this paper we assume a "worst-case scenario" in terms of indexing, meaning that all the candidate items returned by the index are stored in consecutive blocks with 64.5 data items sharing a block on average. This will lessen the effect that false drops have on the retrieval performance, as we do not need a separate page access to fetch each of the false drops.

5.2 Results

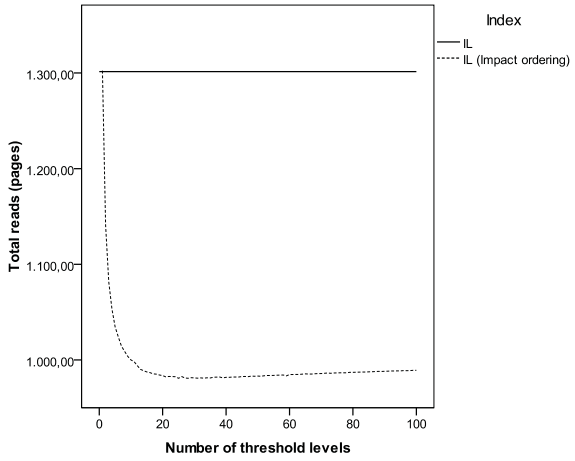
We were interested in several criteria, most importantly in the false drop ratio of the two different approaches and its impact on the query performance. To a lesser extent we also wanted to make sure that the overhead in terms of the index size is not prohibitive. Furthermore, we investigate the impact of the query threshold α on the performance of the indexes. In Sections 5.2.1 to 5.2.3 our competitor is included as a reference; the parameter we vary (number of threshold levels) only applies to our approach in these cases.

5.2.1 False Drop Rate

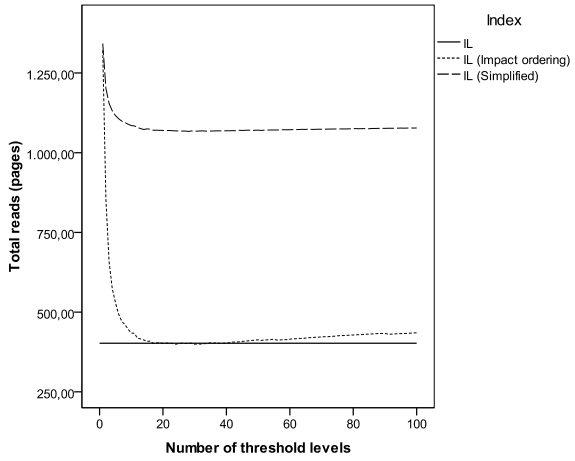
Figure 2 shows the results we obtained for measuring the false drop rate for varying the number of threshold levels in our index (Figure 2(a) depicts the results for possibility measure queries, while Figure 2(b) depicts those for necessity measure queries[6]). Each threshold level covers an equally-sized interval in [0,1]. The false drop rate is measured in the following way: (total number of IDs returned by index - number of true positives)/number of true positives. As can be clearly seen, adding more threshold levels to the data structure reduces the false drop ratio. However, the law of diminishing returns applies to this, i.e. with each added threshold level the improvement gain decreases. We can also notice that using a

[6]The method labeled 'simplified' applies possibility measures as a filter, while the other curve shows the results for the complement-based approach.

ISBN: 978-989-95079-6-8



(a) Possibility measure



(b) Necessity measure

Figure 3: Total number of page accesses

possibility measure as a filter for a necessity measure is not competitive in this scenario.

5.2.2 Retrieval Performance

While the false drop rate suggests to use a very high number of threshold levels, it is not as simple as that to determine the optimal number of threshold levels. By increasing the number of threshold levels we introduce an overhead in terms of index size and retrieval performance (larger indexes result in a longer query processing time). Let us first look at the retrieval performance of the access methods. Figure 3 shows the retrieval performance in total number of page accesses (part (a) for possibility measure queries and part (b) for necessity measure queries). The total number of page accesses includes the page accesses needed to navigate the index and the pages accessed for retrieving all candidate data items (false and true positives).

There is a sharp decrease in the number of page accesses when going from one threshold level to about twenty threshold levels. This is caused by the rapidly dropping false drop ratio. Then the curve peters out and slowly goes up again. In this part of the curve, the drop in performance, due to the aforementioned overhead, gains the upper hand over the slowly decreasing false drop rate. In our benchmark scenario we have identified 25 to be the optimal number of threshold levels for possibility measure queries and 28 to be the optimum for necessity measure queries. Again, the approach using the possibility measure as a filter for necessity measure queries is not competitive, so we will drop it from the graphs in the following sections.

5.2.3 Index Size

In Figure 4 we present the size of the index structures in terms of disk pages. The overhead added for introducing a larger number of threshold levels can be clearly seen. This is caused by a deterioration of the compression ratio. A few larger threshold blocks with smaller gaps between the IDs can be compressed better than a lot of smaller blocks with larger gaps. The stair-like appearance is caused by postings lists expanding by one page at roughly the same time (due to the uniform distribution of domain elements in the possibility distributions). At first glance it seems that our index structure needs more storage space than our competitor. However, as we only need about 20 to 30 threshold levels, this increase is only moderate and, more importantly, we can answer both, possibility measure and necessity measure, queries with the same index. For the other scheme separate indexes are needed (one storing the supports and one storing the cores of the possibility distributions).

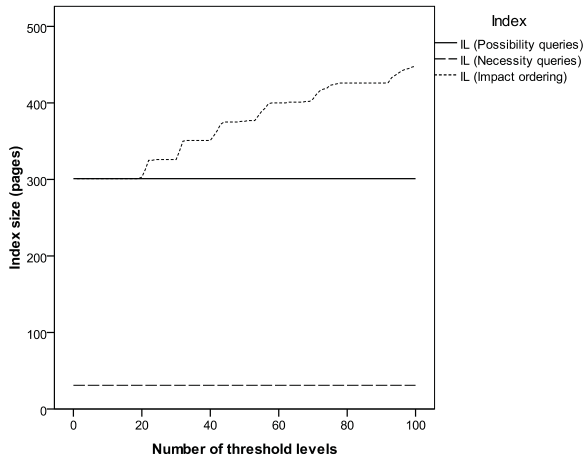
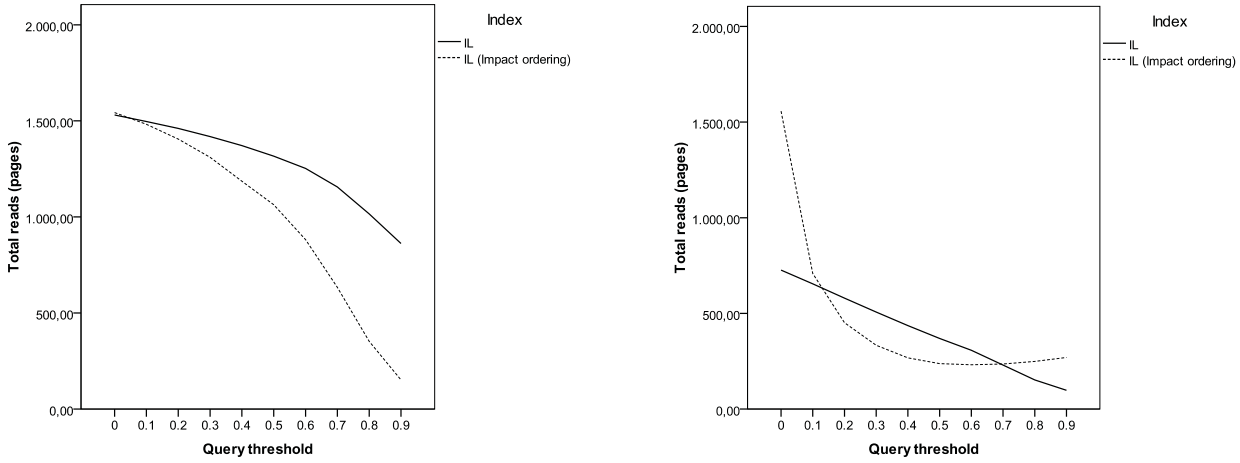


Figure 4: Index size

5.2.4 Impact of Query Threshold

Figure 5 shows the impact of the query threshold (α) on the performance. Generally, as the query threshold grows for possibility measure queries (Figure 5(a)), we have to check smaller and smaller parts of the query set (only elements that have a membership degree larger than or equal to α have to be considered). This in turn decreases the number of postings lists we have to traverse, leading to less page accesses. In addition to this, our approach can stop scanning postings lists earlier for higher query thresholds, leading to further improve-

ISBN: 978-989-95079-6-8



(a) Possibility measure

(b) Necessity measure

Figure 5: Impact of query threshold

ment in terms of the number of page accesses.

For necessity measure queries ((Figure 5(b)), things are more complicated. For the core-filtering approach by Bosc and Galibourg the number of postings lists we have to scan decreases with increasing threshold α. For our approach there are two effects at work. With increasing α the false drop rate drops, which leads to fewer page accesses due to false positives. However, with increasing threshold values the number of page accesses in the index increases. Although the number and lengths of postings list traversals for the possibility measure filter step goes down, the number and lengths of postings list scans for the subsequent complementary check on the necessity measure goes up at quite a fast rate. At some point this effect catches up with us and the number of accessed blocks increases again.

6 Conclusion and Outlook

We have shown how to build an index that stores threshold levels of possibility distributions in a compact way. Being able to represent possibility distributions using a much finer granularity than existing approaches helps us in reducing the number of false positives significantly. Even in an unfavorable environment, our index far outperforms existing approaches for possibility measure queries and was on a par for necessity measure queries. This kind of performance can already be achieved by introducing only a moderate number of threshold levels, meaning that there is practically no overhead, since we only need to maintain one index (while our main competitor needs one for possibility measure queries and one for necessity measure queries).

For future work we would like to analyze how to optimize the number and distribution of threshold levels given a fixed database. Even more interesting would be to optimize and update the index in a dynamic way, given a database with statistics on query workloads.

Acknowledgments

This work has been partially supported by the "Ministerio de Ciencia y Tecnología (MCYT)" (Spain) under grant TIN2007-68084-CO2-01, and the "Consejería de Innovación Ciencia y Empresa de Andalucía" (Spain) under research projects P06-TIC-01570 and P07-TIC-02611. We also thank the anonymous referees for improving the readability of the paper.

References

[1] B. Buckles and F. Petry. A fuzzy model for relational databases. *Fuzzy Sets and Systems*, 7(3):213–226, 1982.

[2] R. Caluwe de, editor. *Fuzzy and Uncertain Object-oriented Databases (Concepts and Models)*, volume 13 of *Advances in Fuzzy Systems - Applications and Theory*. World Scientific, 1997.

[3] R. George, A. Yazici, F.E. Petry, and B.P. Buckles. Uncertainty modeling in object-oriented geographical information systems. In *Third Int. Conf. for Database and Expert System Application (DEXA)*, Valencia, 1992.

[4] F. E. Petry and P. Bosc. *Fuzzy databases: principles and applications*. International Series in Intelligent Technologies. Kluwer Academic Publishers, 1996.

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

[6] J. Galindo, A. Urrutia, and M. Piattini. *Fuzzy Databases: Modeling, Design and Implementation*. Idea Group Publishing, Hershey, PA, USA, 2006.

[7] P. Bosc and M. Galibourg. Indexing principles for a fuzzy database. *Information Systems*, 14(6):493–499, 1989.

[8] C.D. Manning, P. Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[9] C. D. Barranco, J. R. Campaña, and J. M. Medina. A B+-tree based indexing technique for fuzzy numerical data. *Fuzzy Sets and Systems*, 159(12):1431–1449, 2008.

[10] R. Sacks-Davis and J. Zobel. *Indexing Techniques for Advanced Database Systems*. Kluwer Academic Publishers, 1997.

[11] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Morgan Kaufmann, San Francisco, 1999.

[12] A. Trotman. Compressing inverted files. *Information Retrieval*, 6(1):5–19, 2003.

[13] S. Helmer. Evaluating different approaches for indexing fuzzy sets. *Fuzzy Sets and Systems*, 140(1):167–182, 2003.

ISBN: 978-989-95079-6-8