Exploiting the Functionality of Object-Oriented Database Management Systems for Information Retrieval [1]

Gabriele Sonnenberger
Ubilab, Union Bank of Switzerland
Bahnhofstrasse 45, 8021 Zurich, Switzerland
sonnenberger@ubilab.ubs.ch

Abstract

In this paper, we present the approach of FIRE to utilizing an object-oriented Database Management System (DBMS) for Information Retrieval (IR) purposes. First, a comprehensive overview of previous attempts to use DBMSs for implementing IR systems is given. Next, differences between DBMSs and IR systems, with regard to indexing and retrieval, are discussed. In addition, some shortcomings of DBMSs with regard to supporting IR systems are pointed out. Then, an overview of FIRE, which is designed as a reusable IR framework, is given and its approach presented in more detail. Special attention is given to the design and implementation of an IR-index and how retrieval efficiency can be improved by using the optimization facilities of the underlying object-oriented DBMS.

1 Introduction

Due to recent advances in Information Technology and Telecommunications, more and more information is produced and distributed by electronic means. Instead of plain ASCII texts, information is increasingly encoded in different forms, e.g., as graphs, tables or formatted texts. These developments impose additional requirements on the management and retrieval of information. Whereas in the past the focus in Information Retrieval (IR) was on text, which was mostly considered to be unstructured, today's IR systems have to face information which usually consists of structured and unstructured parts and which may be composed of different media.

We observe further changes in the flow and distribution of information. In the past, the user of an IR system was typically a passive consumer of information gathered at the special sites of professional information providers; while nowadays a user is often both an information consumer *and* an information provider, e.g., in a cooperation's document management system. Hence, data management issues like persistent storage of data, concurrency control, and recovery after failures are gaining importance, and IR systems must cope with these issues.

Developing techniques for the indexing and retrieval of heterogeneous information units is a demanding topic genuine to the IR domain. In contrast to this, data management issues have already been investigated extensively by the Database (DB) community, which has also developed theories and techniques applied in productive systems. Rather than 'reinventing the wheel', it is natural for developers of IR systems to try to profit from the results of the DB community by basing IR systems on Database Management Systems.

[1] Appeared in: IEEE Data Engineering Bulletin, Vol. 19, No. 1, pp. 14-23, 1996

In this paper, we report on our experience utilizing a Database Management System (DBMS) for IR purposes. Section 2 reviews previous attempts to use DBMSs for implementing IR systems. In Section 3, we discuss differences between IR systems and DBMSs with regard to indexing and retrieval, and point out some shortcomings of DBMSs with regard to supporting IR systems. Finally in Section 4, we sketch our IR framework, which we call FIRE, and present our approach for using the functionality of an object-oriented DBMS in an IR system. The paper concludes with some remarks on future work.

2 IR Systems Based on DBMSs

2.1 Relational DBMSs

A first approach to use a DBMS for IR purposes is to consider IR as a DB application. This has been proposed, for instance, by Macleod & Crawford [Mac83], Blair [Bla88], and Smeaton [Sme90]. Common to these proposals is that the IR systems are based on a relational DBMS, and SQL is used as retrieval language. These IR systems do not store full documents, but bibliographic references to documents; these usually provide the title of the document represented, the names of the authors, an abstract, some content descriptors, and the details about the date and location of the publication.

This approach of making use of a DBMS has received major criticisms. One factor that has been criticized especially, e.g., by Schek & Pistor [Sch82], is that the relational DB model represents information in a rather unnatural way by a set of tables (relations). Further, SQL queries tend to be rather complex and difficult to understand; see for instance the examples given in [Mac91]. In addition, retrieval may be quite computationally expensive, especially when information from different tables has to be combined by a 'join' operation.

A severe shortcoming of this approach is the restriction to reference retrieval. This restriction is not so much voluntary as a matter of the underlying relational DB model. The relational model requires the fields of a record to be of a fixed length, thus making it difficult for a relational DBMS application to manage information units such as ordinary texts, which usually vary in length. (Of course, there are work-arounds like follow-up records, but such work-arounds make the modeling of information even more awkward.) A second severe shortcoming is that SQL in its basic form is restricted to exact matching. An exact match is appropriate for many DB applications, especially when information is structured to a high degree and the vocabulary used is rather fixed. In most IR applications however, information units like texts are significantly less structured and the vocabulary used is usually unrestricted. Correspondingly, users of IR systems find it far more difficult or even impossible to issue a query which successfully delivers all information relevant to a given information need, but excludes irrelevant material. Therefore, more advanced approaches to IR abandoned the exact matching paradigm and find instead the pieces of information which *best match* the user's query by applying weighting schemata. The user receives as result a *ranked list* which is sorted by the assumed probability that a piece of information is suited to answer the user's information need.

Several attempts have been made to provide better DB support for the development of IR systems by extending the relational model, or by adding new features to SQL. To provide more natural external views of information, Schek & Pistor [Sch82] have proposed a generalization of the relational model which allows nested relations. Lynch & Stonebraker [Lyn88] have introduced abstract data types, which make the formulation of content-based search conditions more convenient, although retrieval is still based on the exact matching paradigm. An extension of SQL allowing a 'similar-to' comparison operator has been proposed by Motro [Mot88]. Furthermore, various approaches have been developed for dealing with uncertain information, for instance by Garcia-Molina & Porter [Gar90].

2.2 Coupling an IR System with a DBMS

A different approach to making use of a DBMS for IR purposes is to couple an IR system with a DBMS. Croft et al. [Cro92] attempted a loose coupling between the IR system INQUERY and IRIS, a prototype version of an object-oriented DBMS. They chose an object-oriented DBMS since the object-oriented model allows one, in contrast to the relational model, to store complex textual information in a quite natural way. The IR and the DB system are coupled externally by a control module. There are links from the information units stored by INQUERY to the corresponding textual objects of the IRIS database. The integrated system provides the functionality of both underlying systems. Thus, the user may issue content-based queries as well as DB queries. However, a severe problem of this coupling approach is that information is stored twice, by the IR system as well as by the DBMS. This may cause consistency problems when information is modified. Furthermore, there is no full integration of content-based queries and DB queries.

Gu et al. [Gu93] have chosen an alternative way to couple an IR system with a DBMS. They have embedded the functionality of the IR system INQUERY into the relational DBMS Sybase. Furthermore, they have extended SQL by a function which takes an INQUERY query as input and allows one to choose the INQUERY database to be consulted for evaluating the query. This system avoids storing information twice: textual information is stored and managed by the IR system and other kinds of information by the DBMS, which also provides pointers to the textual information stored and managed by the IR system. The main drawback is that two different approaches for managing and retrieving information are used, which makes the management of information more difficult. Further, the best match retrieval paradigm is restricted to textual information, whereas an application of this retrieval paradigm to other kinds of information would be highly desirable, as for instance pointed out by Fuhr [Fuh92].

2.3 Object-oriented DBMSs

Bearing in mind the shortcomings of the relational model, it has been proposed to use other models, more appropriate than the relational DB model, as a basis for developing IR systems; e.g., an array model [Mac87] or an object-oriented DB model [Har92p]. The object-oriented DB model is especially appealing, since object-oriented technology is maturing and the first commercial systems are already available.

The object-oriented approach has also been adopted for developing our IR framework FIRE. The framework is implemented using ObjectStore [Lam91], a commercially available object-oriented DBMS. In ObjectStore, persistence is not part of the definition of an object, but a matter of allocation at the time of object creation ('persistency by allocation'). Thus, objects of the same type can be allocated persistently as well as transiently. Furthermore, it makes essentially no difference whether we deal with a transient or persistent object. These features of ObjectStore enable us to design and implement an IR system in a problem-adequate way, mostly neglecting data storage details. Thus the developer of an IR system can rely on the DBMS's means for persistent storage, concurrency control, recovery after failure, etc., without having to accept severe restrictions, as experienced from relational DBMSs.

Object-oriented DBMSs provide a useful basis for the development of IR systems, yet could support IR systems even further with regard to performance issues. ObjectStore, for instance, allows one to improve retrieval efficiency by building specialized indexes and by optimizing queries. However, indexing and retrieval in an IR system differ in certain aspects from indexing and retrieval in a DBMS, as described in the following section. Consequently these facilities cannot be used directly for IR purposes. Modifying the core functionality of a DBMS for exploiting these facilities is not in the range of a regular user of a DBMS, thus we cannot expect a solution from the DB side — at least not in the short term. Yet, the optimization facilities of object-oriented DBMSs can be utilized in IR systems by a proper design on the IR side, as is shown in Section 4.

3 Some Differences between IR Systems and DBMSs

3.1 Index

The term 'index' is used both by the DB community and by the IR community, however with different meanings. While the function of a DB-index is to improve performance, an IR-index typically also provides additional data. In detail, we can observe the following differences:

- A typical IR-index is an inverted file with index entries, each consisting of a key representing a feature derived from a source and a set of postings. A posting may include information about the frequency of the given feature within the source, or may specify its position within the document in more detail. A DB-index, however, only maintains the paths to the objects where an attribute has a certain value.
- In an IR application, information about the characteristics of the collection represented by an index is required when computing the relevance of an information unit to a user's query. An example of such information is the maximum frequency of a term in a collection of texts or the mean value and standard deviation of a set of numbers. Since a DB-index serves only to optimize access, no such collection information is needed.

Due to the differences, typical IR-indexes are not supported by DBMSs. Optimized access to information — as is done by a DB-index — is, however, crucial for implementing efficient IR systems. Thus, how we can use DB-indexes to build IR-indexes is an open research problem.

3.2 Retrieval

A typical DBMS provides a retrieval interface which allows one to check whether two values are equal, and to determine whether a value is smaller or greater than another value. Usually such queries can be optimized, e.g., by instructing the DBMS to create and maintain appropriate indexes.

Advanced IR systems are supposed to support approximate ('best') matching. Unfortunately, approximate matching is not supported by DBMSs. This is a severe shortcoming, since approximate matching is far more time-consuming than exact matching, and therefore demands optimization facilities. To deal with the additional complexity, we urgently need specially tuned matching methods and techniques to avoid a linear search through an index. These methods and techniques should ideally work hand in hand with available DBMS techniques.

3.3 Indexing and Retrieval Functionality

In the case of a conventional IR application, i.e. one which is not based on a DBMS, the application developer has to provide all the indexing and retrieval functionality needed. Developing an IR system on top of a DBMS might save design and implementation effort, since DBMSs already provide indexing and retrieval functionality. The task of the application developer would then be to choose appropriate functionalities and to instruct the DBMS correspondingly. In order to index a set of objects for instance, the application developer would have to state which object attributes are to be indexed in which way, and to select the index structures best suited for the attribute values. Indexes would then be set up automatically under control of the DMBS. Unfortunately, the options provided by DBMSs do not cover the full range needed for IR applications and do not allow one, for instance, to index a text directly in a traditional IR manner using the indexing mechanisms of a DBMS. Thus, we need to investigate how we can use indexing and retrieval functionalities of object-oriented DBMSs for IR purposes.

4 The Approach of FIRE

4.1 An Overview of FIRE

FIRE is designed to facilitate the development of IR applications and to support the experimental evaluation of indexing and retrieval techniques. In addition, the framework is supposed to provide basic functionalities for several media and should end up as a flexible and extensible tool that can be used for developing a wide range of IR applications. FIRE is an acronym for '**F**ramework for **I**nformation **Re**trieval Applications'. It is developed in cooperation with the IR group at the Robert Gordon University, U.K.

The design and implementation of FIRE are based on an object-oriented approach. The object model of FIRE defines the basic IR concepts and supplies functionalities that support the realization of an IR application. This section gives a short overview of FIRE focusing on the most essential classes of FIRE's object model. A more comprehensive description of FIRE's design and object model is given in [Son95].

FIRE represents documents by a set of features (or attributes). Note that the term document is used here in a broad sense: documents may consist of structured and unstructured parts and may be composed of different media. The class *ReprInfoUnit* (see Figure 1) models documents in a generic way. It defines methods which provide information about the modeling of a document representation as well as methods for accessing the features of a document representation. In addition, *ReprInfoUnit* and its subclasses are responsible for organizing the indexing and retrieval of documents of the respective type.

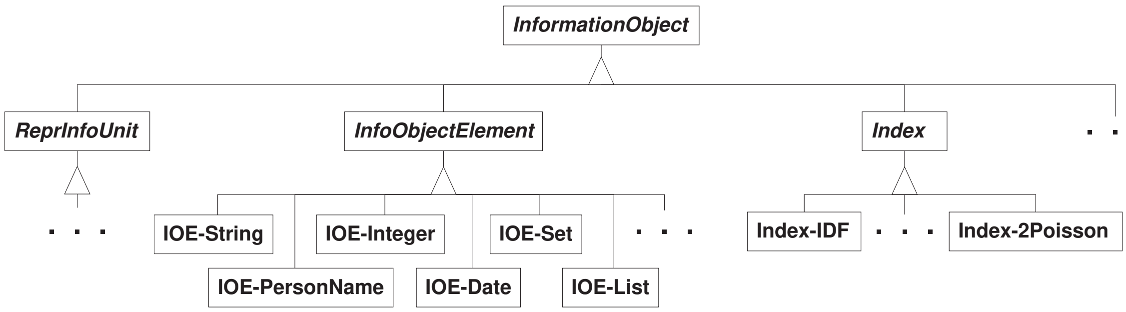

Figure 1: Overview of the most important classes of FIRE's object model

Concrete subclasses of *ReprInfoUnit* define how documents of a certain type, e.g., books, tables, etc., are represented in an application. When dealing with text for instance, a new subclass of *ReprInfoUnit* may be introduced, consisting of features like *Title*, *Authors*, *TextBody*, and *PublicationDate*. The modeling of concrete types of documents is not part of FIRE's object model as this is an application-specific task. Nevertheless, the framework supports the application developer in this task. The class *InfoObjectElement* and its subclasses provide a set of data types like string, integer, person name, date, set, and list (c.f. Figure 1), which are intended to be used for the application-specific modeling of documents. If needed, the application developer may extend this set of data types by adding new subclasses. *InfoObjectElement* helps to reduce the effort for developing an IR application by providing an associated interface for indexing an information unit, determining the similarity of two units, etc. Further, this branch of the class hierarchy supports a uniform representation of document components, e.g., the author of a book is specified in the same way as the author of a chart table.

The class *Index* is the implementation of a typical IR-index, which not only manages a set of indexing features derived from a collection of documents but also provides information for calculating the probability of relevance of information units to the user's query. The class *Index* is a generic class, which solves general tasks but does not provide specific IR functionality. The latter is supplied by concrete subclasses of *Index* like *Index-IDF* and *Index-2Poisson*, which implement particular weighting schemata (see [Har92m] for an overview of weighting schemata).

In the following two sections, we discuss the design and implementation of an IR-index in FIRE more comprehensively. First, we present the design of the class *Index* with focus on IR related issues. Then, we discuss some details of *Index* which support an efficient evaluation of queries and enable us to exploit the optimization facilities of the underlying DBMS.

4.2 Design of an IR-Index

An *Index* basically consists of a set of *IndexingFeature*s (see Figure 2). In addition, it may be associated with zero or more objects of the type *DB-Collection*, whose purpose is explained in the next section. An *IndexingFeature* consists of a feature, e.g., a normalized word from a text, and a source specification. The latter is essentially a reference to the document from which the feature has been derived. In addition, an *IndexingFeature* may specify a position within the source. Positional information is needed for indicating to the user why a particular information unit has been retrieved, which is done by 'highlighting' the relevant pieces of the unit. Furthermore, weighting schemata may use positional information, e.g., it may be assumed that words occurring in a heading are more important than words in regular paragraphs. Finally, an *IndexingFeature* is associated with the indexing method that has been used for deriving the feature. Information units may be indexed in many different ways, and in many cases more than one method can be applied for indexing a particular unit, e.g., weak or strong stemming when indexing textual units. For consistency reasons, it is important that all *IndexingFeature*s of an *Index* have been derived by the same method. Hence, FIRE associates *IndexingFeature*s with the indexing method applied, and checks whether the *IndexingFeature* to be added to an *Index* is compatible with the ones previously added.

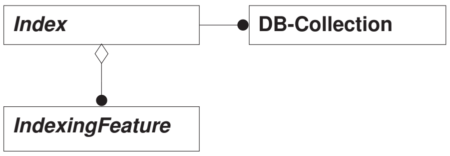

Figure 2: Structure of the class *Index*

The class *Index* defines a set of operations and methods (see Figure 3), which provide a uniform interface independent of any particular retrieval model. The method *addIFs* of *Index* incorporates a set of *IndexingFeature*s into an *Index*. The adding of features does not cause any updating activities like sorting the *Index* or re-calculating indexing weights. Updates have to be invoked explicitly by an *update* message. We chose to separate the processes in order to avoid unnecessary computations, for instance, sorting an *Index* again and again when indexing a whole collection of documents. The class *Index* also provides a method, called *getIFsOf*, for determining the *IndexingFeature*s which have been derived from a particular source. By the method *retractIFsOf*, the *IndexingFeature*s referring to a given set of sources can be retracted, whereas the method *clear* removes all *IndexingFeature*s from an *Index*. Finally, the method *retrieve* serves for retrieving information by evaluating single query conditions. The results of the evaluation are passed to the query document, which is a *ReprInfoUnit* object. The query document invokes appropriate methods for combining the results and for computing the scores ('Retrieval Status Values', RSVs) indicating the estimated relevance of an information unit to the user's query.

Index
addIFs(features: IndexingFeatures): Boolean *update(): Boolean* *getIFsOf(source: IO-Address): IndexingFeatures* *retractIFsOf(sources: IO-Addresses): Boolean* *clear(): Boolean* *retrieve(condition: QueryCondition): BasicRetrievalResults* *supportMatchers(names: Strings): Boolean* *getSupportedMatchers(): Strings*

Figure 3: Operations and methods of *Index*

FIRE supports an approximate matching on different data types. It allows one, for instance, to retrieve documents which cover a particular topic with a high probability and to retrieve documents which have a similar author name or a similar publication date. As discussed before, an approximate matching may be quite time consuming. In order to compensate computing efforts, an *Index* object may be instructed to support particular matching methods. This can be done by an authorized user at run-time via the method *supportMatchers*. Also, previous instructions may be overwritten by new ones. Note that an *Index* always allows one to use any matching method applicable to the given type of *IndexingFeatures*, but performs supported methods more efficiently. The details of the optimization of the matching process are fully encapsulated by the class *Index*. This is advantageous as it avoids bothering the user with optimization details.

The subclasses of *Index* implement particular weighting schemata. They define additional methods which calculate the information required by the respective weighting schema. Figure 4 shows an example subclass of *Index* which implements a version of the 'Inverse Document Frequency' (IDF) weighting schema.

Index-IDF
getNumberOfSources(): Integer getMaxFreqOfAnyIndexingFeature(): Integer getMaxFreqOfIndexingFeature(f: IndexingFeature): Integer getFreqOfIndexingFeature(f: IndexingFeature, s: IO-Address): Integer getIndexingWeightOfIndexingFeature(f: IndexingFeature, s: IO-Address): Real

Figure 4: A concrete subclass of *Index*

The subclasses of *Index* are not specialized to a particular type of indexing features. Hence, they can be used in different contexts and the application developer needs to define a new subclass only if an additional weighting schema is to be supported.

4.3 Improving Retrieval Efficiency

An *Index* may be associated with zero or more objects of the class *DB-Collection* (see Figure 2). A *DB-Collection* serves to improve retrieval efficiency. It exploits the optimization facilities of the DBMS and may support approximate matching methods.

An object of the class *DB-Collection* consists of a collection of *IndexEntry*s. An *IndexEntry* is composed of a key, which may be of any type, and a set of pointers to the *IndexingFeature*s of an *Index* which yield the same key. (Thus, *IndexEntry*s have essentially the same structure as entries of an inverted file.) The key of an *IndexEntry* is derived from the corresponding *IndexingFeature*, and may have the same value as the feature or may be assigned some code for optimizing access. The interface of *DB-Collection*, which is depicted in Figure 5, is similar to the interface of *Index*. However, *DB-Collection* is internally concerned with *IndexEntry*s rather than *IndexingFeature*s. The derivation of *IndexEntrys* from *IndexingFeatures* is invoked by *DB-Collection*.

DB-Collection
addIFs(features: IndexingFeatures): Boolean update(): Boolean retractIFs(features: IndexingFeatures): Boolean clear(): Boolean xMatch(feature: IndexingFeature): IndexingFeatures setMatcher(name: String): Boolean getMatcher(): String setOptionsDB-Index(options: String): Boolean getOptionsDB-Index(): String

Figure 5: Interface of *DB-Collection*

A *DB-Collection* utilizes the query optimization facilities of the underlying DBMS by instructing the DBMS to create an appropriate DB-index for the given collection of *IndexEntrys*. Since the DBMS does not index document representations but collections of *IndexEntrys*, we can apply IR indexing techniques without being restricted in any way by the DBMS. Nevertheless, we can take advantage of the DBMS's facilities for optimizing the evaluation of queries. As a further advantage, FIRE does not need to provide any specialized index structures (B-trees, hash tables, etc.) as well as query optimization strategies, but can rely on ObjectStore's means.

In addition, *DB-Collection* serves to optimize approximate matching. When we try to find the objects of a collection which are similar to a given object, we have to consider all objects. This is in contrast to an exact matching where search can be restricted in most cases, e.g., by a binary search in an ordered index. To reduce complexity, FIRE allows one to perform the approximate matching in a *restricted* form, which is a combination of an exact and an approximate matching. In this matching mode, a key is generated in a first step for the *IndexingFeature* given with the query condition. Such a key may be for instance a phonetic code for a person name. Then the corresponding *DB-Collection* is consulted and its *IndexEntrys* looked-up to determine the *IndexingFeatures* for which the same key has been generated. Finally, a full approximate matching is performed with the selected *IndexingFeatures*. Note, an *Index* may support more then one approximate matching method at the same time by creating different *DB-Collections*. This is useful, since different retrieval situations may require different matching methods. In the case no appropriate *DB-Collection* exists for the matching method to be performed, the *Index* does the look-up itself, of course in a less efficient way, by going through the associated set of *IndexingFeatures*.

The derivation of keys from *IndexingFeatures* is a matter for the matching algorithms, since they know best how to build appropriate keys. Also, the matching algorithms determine which kind of DB-index is most appropriate for the resulting keys. In FIRE, matching algorithms are represented by specialized classes rather than by methods of other classes. This design decision allows the user to browse through the class hierarchy in order to see which matching algorithms are available and how they are to be used; see [Son96] for further details. The classes implementing particular matching algorithms are subclasses of an abstract class called *Matcher*, which is depicted in Figure 6. The important features of this class with regard to the current topic are the method *key* for deriving keys from *IndexingFeatures* (or *InfoObjectElements*) and the attribute *OptionsDB-Collection* for specifying the DB-index options to be applied for supporting a particular matcher.

Matcher
TypeInfoObjectElement: String *TypeDB-Collection: String* *OptionsDB-Collection: String*
key(o: InfoObjectElement): Object *key(o: IndexingFeature): Object* *match(o1: InfoObjectElement, o2: InfoObjectElement): Real* *match(o1: IndexingFeature, o2: IndexingFeature): Real* *keyMatch(o1: InfoObjectElement, key1: Object, o2: InfoObjectElement): Real* *keyMatch(o1: IndexingFeature, key1: Object, o2: IndexingFeature): Real* *restrictedKeyMatch(o1: IndexingFeature, key1: Object, o2: IndexingFeature, r: IO-Addresses): Real*

Figure 6: Class *Matcher*

The design of the class *Matcher* and its subclasses eases the optimization of approximate matching methods. It is fully sufficient to instruct an *Index* to support certain *Matcher*(s). Knowing the names of the matching algorithms to be supported, the *Index* itself can gather the necessary details by asking the proper *Matcher*(s).

Conclusions

In this paper, we have reviewed previous attempts to use DBMSs for implementing IR systems and pointed out some shortcomings of DBMSs with regard to the support of IR systems. Furthermore, we have presented FIRE's approach for using the functionality of an object-oriented DBMS in an IR system. FIRE allows one especially

- to represent complex heterogeneous information in a natural way,
- to utilize the query optimization facilities of the underlying DBMS for IR purposes, and
- to reduce the complexity of an approximate matching.

In the near future, we will perform experiments to quantify the effect of the DBMS's optimization facilities in IR applications and to test the performance of the restricted approximate matching.

Acknowledgements

The author would like to thank Tore Bratvold, Ubilab, and Andreas Geppert from the University Zurich for many fruitful discussions on the design of FIRE and the integration of DBMSs and IR systems. Thanks are also due to Manuel Bleichenbacher, Ubilab, for his work on ETOS, which is the software development platform used for implementing FIRE, and his technical assistance with ETOS.

References

[Bla88] D.C. Blair: An Extended Relational Document Retrieval Model. In: *Information Processing & Management*, Vol. 24 (3), pp. 349-371, 1988

[Cro92] W.B. Croft, L.A. Smith, and H.R. Turtle: A Loosely-Coupled Integration of a Text Retrieval System and an Object-Oriented Database System. In: *Proc. of the 15th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR-92)*, pp. 223-232, 1992

[Fuh92] N. Fuhr: Integration of Probabilistic Fact and Text Retrieval. In: *Proc. of the 15th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR-92)*, pp. 211-222, 1992

[Gar90] M. Garcia-Molina and D. Porter: Supporting Probabilistic Data in a Relational System. In: *Proc. of EDBT*, pp. 60-74, 1990

[Gu93] J. Gu, U. Thiel, and J. Zhao: Efficient Retrieval of Complex Objects: Query Processing in a Hybrid DB and IR System. In: G. Knorz, J. Krause and C. Womser-Hacker (eds.): *Information Retrieval '93, Von der Modellierung zur Anwendung*, pp. 67-81, Konstanz/Germany: UVK, 1993

[Har92m] D. Harman: Ranking Algorithms. In: W.B. Frakes and R. Baeza-Yates (eds.): *Information Retrieval: Data Structures & Algorithms*, Englewood Cliffs/N.J.: Prentice Hall, 1992

[Har92p] D.J. Harper and A.D.M. Walker: ECLAIR: An Extensible Class Library for Information Retrieval. In: *The Computer Journal*, Vol. 35 (3), pp. 256-267, 1992

[Lam91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb: The ObjectStore Database System. In: Communications of the ACM, Vol. 34 (10), pp. 50-63, 1991

[Lyn88] C.A. Lynch and M. Stonebraker: Extended User-Defined Indexing with Applications to Textual Databases. In: Proc. of the 14th *International Conference on Very Large Data Bases*, pp. 306-317, 1988

[Mac83] I.A. Macleod and R.G. Crawford: Document Retrieval as a Database Application. In: *Information Technology: Research and Development*, Vol. 2, pp. 43-60, 1983

[Mac87] I.A. Macleod and A.R. Reuber: The Array Model: A Conceptual Modeling Approach to Document Retrieval. In: *Journal of the American Society for Information Science*, Vol. 38 (3), pp. 162-170, 1987

[Mac91] I.A. Macleod: Text Retrieval and the Relational Model. In: *Journal of the American Society for Information Science*, Vol. 42 (3), pp. 155-165, 1991

[Mot88] A. Motro: VAGUE: A User Interface to Relational Databases that Permits Vague Queries. In: *ACM Transactions of Office Information Systems*, Vol. 6 (3), pp. 187-214, 1988

[Sch82] H.J. Schek and P. Pistor: Data Structures for an Integrated Database Management and Information Retrieval System. In: *Proc. of the 8th International Conference on Very Large Data Bases*, pp. 197-207, 1982

[Sme90] A. Smeaton: Retriev: An Information Retrieval System Implemented on Top of a Relational Database. In: *Program*, Vol. 24 (1), pp. 21-32, 1990

[Son95] G. Sonnenberger and H.-P. Frei: Design of a Reusable IR Framework. In: *Proc. of the 18th Annual Int. ACM SIGIR Conference on R & D in Information Retrieval (SIGIR '95)*, pp. 49-57, 1995

[Son96] G. Sonnenberger, T.A. Bratvold, and H.-P. Frei: Use and Reuse of Indexing and Retrieval Functionality in a Multimedia IR Framework. (Paper presented at the final MIRO Workshop in Glasgow, Sept. 1995; publication forthcoming)