Integrating DB and IR Technologies: What is the Sound of One Hand Clapping? *

Surajit Chaudhuri[1], Raghu Ramakrishnan[2], Gerhard Weikum[3]
1) Microsoft Research, Redmond, WA 98052, USA (surajitc@microsoft.com)
2) University of Wisconsin, Madison, WI, 53706, USA (raghu@cs.wisc.edu)
3) Max-Planck Institute of Computer Science, D-66123 Saarbruecken, Germany (weikum@mpi-sb.mpg.de)

Abstract

Databases (DB) and information retrieval (IR) have evolved as separate fields. However, modern applications such as customer support, health care, and digital libraries require capabilities for both data and text management. In such settings, traditional DB queries, in SQL or XQuery, are not flexible enough to handle application-specific scoring and ranking. IR systems, on the other hand, lack efficient support for handling structured parts of the data and metadata, and do not give the application developer adequate control over the ranking function. This paper analyzes the requirements of advanced text- and data-rich applications for an integrated platform. The core functionality must be manageable, and the API should be easy to program against. A particularly important issue that we highlight is how to reconcile flexibility in scoring and ranking models with optimizability, in order to accommodate a wide variety of target applications efficiently. We discuss whether such a system needs to be designed from scratch, or can be incrementally built on top of existing architectures. The results of our analyses are cast into a series of challenges to the DB and IR communities.

1. Introduction

1.1 Motivation and State of the Art

DB and IR systems are currently separate technologies. Thirty years ago, the application classes that drove the progress of these systems were disjoint and did indeed pose very different requirements: classical business applications like payroll or inventory management on the DB side, and abstracts of publications or patents on the IR side. The situation is radically different today. Virtually all advanced applications need both structured data and text documents, and information fusion is a central issue. Seamless integration of structured data and text is at the top of the wish lists of many enterprises. Example applications that would benefit include the following:

- Customer support systems that track complaints and response threads and must ideally be able to automatically identify similar earlier requests.
- Health care systems that have access to the electronic information produced by all hospitals, medical labs, and physicians (with appropriate measures for privacy and authorization), and have the capability of monitoring trends and generating early alerts about health crises such as epidemic diseases.
- Intranet search engines for organizations with richly structured data sources in addition to traditional document repositories.

There are commercial systems that address the above applications and provide partial solutions. A typical state-of-the-art data-plus-text system is a collection of platforms and tools that are "glued together" in an ad-hoc, application-specific manner. The complexity that an application developer faces when using such systems is tremendous, yet they often lack options needed by advanced developers. Integration of database (DB) and information retrieval (IR) technologies has been listed among the major challenges of our field in [Lo03]. Throughout this paper, we take the viewpoint of the developer of an advanced application, such as those mentioned above, who wants to gain productivity by building on top of a generic platform, but also needs explicit control over application-specific details, especially scoring and ranking. We also focus on retrieval and query capabilities, although requirements for features like concurrency and recovery often interact with the design of query-related features. With this in mind, the currently available commercial systems may be classified into the following three categories:

- ***DB+IR:*** There are a few niche-market systems that have been built from scratch for specific application

Proceedings of the 2005 CIDR Conference

---

* This is a famous koan, one of the cryptic riddles of Zen Buddhism that a master may give to his student as a meditation exercise. The analogy is that the answer would be truly enlightening, but it is unfortunately unclear if there really is an answer.

classes with integrated text and data management. A good example is the QUIQ system [Ka03] for customer support applications. It takes a unified approach to handling structured data and text fields, including scoring for similarity search, but, it was not designed to be a universal infrastructure platform, and its functionality cannot be easily extended to handle applications beyond its originally intended target domain.

- ***IR:*** The market-leading intranet and Web search engines (e.g., Verity, Inktomi, Google) provide rich functionality and flexibility for text search, including powerful options for thesauri, feature extraction, and alternative scoring methods. Although their heritage is document-oriented, these systems have recently paid more attention to structured data and provide some XML support as well. Widely unknown to DB folks is that intranet search engines come with a rich API, and are not just end-user-oriented platforms. However, this class of systems lacks the query optimization that database engines have. Thus, the programmer also has the burden of making decisions about search strategies (e.g., evaluation orders) or must rely on default behavior with "suboptimal" performance. This issue becomes critical when applications need joins and more complex queries, especially when these access both text and structured data.
- ***DB:*** Most commercial database systems (e.g., Oracle, IBM DB2, Microsoft SQL Server) have a text extension that handles search on text documents. Typically, the text extension is a separate engine that is not really integrated with the database engine. At the API level, search predicates on text follow different concepts and syntax compared to SQL, and sometimes specific syntax is required to guide the query processor on index usage and other optimizations that should ideally be user-invisible if the system truly provided data independence. Last but not least, despite the fact that these systems are promoted as extensible database technology, they provide hardly any flexibility in their scoring and ranking functionality. Although such limited systems have found use in many applications that combine data and text, we must recognize that building, for example, a versatile, customizable intranet search system on top of such a platform is virtually impossible.

Thus, both DB and IR systems have critical gaps that the other side fills, and much functionality is currently glued-on in a way that poses undue complexity to application developers. In summary, there is insufficient text support in the DB world, and no query optimization for advanced queries in the IR world. What is badly needed is a platform that integrates DB and IR technologies, is open and flexible with regard to scoring and ranking, and yet is easy to use by a reasonably experienced programmer. In addition, there must be an underlying query optimizer that is aware of ranking methods.

To conclude this motivating discussion, we have to admit that despite our critical remarks about existing systems it is very difficult to assess the benefits and idiosyncrasies of each system, leaving us with:

Challenge 1: How should we systematically compare different systems with partial DB&IR support in terms of ease of use, flexibility and versatility, search result quality, and efficiency? What is an appropriate experimental framework for such a comparison?[*]

1.2 Contributions and Paper Organization

This paper takes a new look at the old problem of how to integrate structured data and text with similarity search based on scoring and ranking. We aim at a deeper understanding of requirements and possible system architectures. In doing this we have tried to reconcile the following three goals: 1) Providing rich functionality to advanced developers in an easy-to-use API. 2) Providing a means of extending and customizing scoring and ranking methods. 3) Enabling the database engine's query optimizer to effectively handle scoring and ranking.

The paper is organized as follows:

- In Section 2, we discuss three motivating application scenarios, and derive a set of requirements for a core DB&IR system.
- In Section 3, we consider architectural issues in integrating text-processing capabilities with database query languages, and argue for a RISC-style addition of a DB&IR layer to a simplified database system.
- In Section 4, we consider possible foundations for the DB&IR layer, and present several alternative extensions of relational algebra that support flexible ranking and scoring, and allow for early termination after computing the most highly ranked results.
- In Section 5, we illustrate the major challenges in query optimization and evaluation by discussing the limitations of our algebras with respect to the set of algebraic equivalences that they (mostly do *not!*) allow. Collectively, Sections 4 and 5 are intended to highlight the main challenges through discussion of technical specifics; we make no claims that the algebras we propose are most appropriate.

1.3 Related Work

Ranked retrieval of data has been studied in the IR community for over 30 years [BR99, WMB99], and in the DB field for at least 10 years with an emphasis on multimedia data and approximate query processing [GJM97]. However, only a small subset of this prior work has been concerned with systematically reducing a variety of methods onto a compact yet versatile core set of functions, and even fewer papers have tried to integrate such a core with

---

[*] Note that the TREC and INEX benchmarks in IR focus on search result quality alone. Only INEX considers semistructured data, but is still very much document-centric and hardly addresses structured data aspects.

existing technology for query execution and optimization. Most notably, recent work on score and rank aggregation [NR99, GBK00, Na01, FKS03, Fa03, FLN03, BGM02, CH02, YPM03] has led to some convergence in the computational models, but integration of ranked retrieval into a DB engine remains open; for example, it is unclear how Fagin's threshold algorithm (TA) could be incorporated into a state-of-the-art query processor and optimizer.

IR work is typically (though not exclusively) more concerned with effectiveness (i.e., search result quality, under metrics such as precision and recall) than with efficiency. For example, the TREC and INEX benchmarks do not consider response time or throughput. In terms of effectiveness, IR has made major progress in the last few years by developing methods based on statistical language models and other machine learning and web mining techniques [CL02, MS99, Cha02]. The most striking "weakness" of IR technology, in our opinion, is the absence of automatic query optimization for advanced queries.

The most relevant lines of work on how to combine top-k similarity queries with query processing technology is by Chaudhuri et al. [CDY95], who investigated execution strategies for database queries with external text sources, Bruno et al. [BCG02], who mapped top-k queries onto multidimensional range queries, Carey and Kossmann [CK97, CK98], who introduced a stopping operator into pipelined query execution plans, Donjerkovic and Ramakrishnan [DR99], who discussed a probabilistic optimization framework for top-k queries, Kießling et al. [Ki02, LK02] and Hristides et al. [HKP01], who developed models and systems for specifying user preferences in queries (e.g., on product catalogs), Cohen [Co98, Co00], Agrawal et al. [Ag03], and Ilyas et al. [IAE03] who adopted and extended IR-style statistical measures for ranking of SQL query results, and Kabra et al. [Ka03] on the specific ranking and query processing model of the hybrid DB&IR system QUIQ. Very recently, Ilyas et al. [Il04] and Theobald et al. [TWS04] have looked into estimating the distributions of scores in ranked lists for query rewriting and run-time optimizations.

From an IR viewpoint, the efficient processing of inverted index lists and models for applying probabilistic IR to structured data are the issues that are closest to database query processing. Relevant research on index processing includes the work by Moffat and Zobel [MZ96], who developed heuristics for pruning the scanning of long inverted lists while maintaining retrieval quality, by Long and Suel [LS03] as well as Brin and Page [BP98], who develop additional heuristics as well as TA-style robust pruning techniques, Fuhr et al. [FGA03] who have extended these approaches to an XML setting, and Kaushik et al. [Ka04], who discuss how to combine inverted lists and path indexes for XML, and generalize Fagin's TA framework in the presence of indexes, as is the case within a core DB engine.

Relevant research on probabilistic IR for structured data includes the seminal work by Fuhr [Fu00] on probabilistic Datalog and W. Cohen's WHIRL system [Co98, Co00], and recent work presented in [ACD02, As02, Ch04, HP02, BHP04]. An overview of other IR approaches to handle structured data is given in [BR99]. None of this work addresses full-fledged compositionality of operators with ranked results, in combination with early termination as in top-k index processing.

There is increasing interest in studying ranked retrieval on semistructured XML data, a topic that implicitly calls for combining DB and IR technologies, in both communities,. This includes work on structural similarity search on XML trees, using, e.g., tree edit distance measures [SM02, ACS02, ZAR03], work on text pattern matching tailored to XML notions of adjacency [Am03, AYJ03, Ba03, FKM00], and work that generalized IR similarity measures using word statistics, thesauri, etc. [Co03, CK02, Ca03, FG04, GBS01, Gu03, SM02, STW03, TW00, TW02]. Most of this prior and ongoing research focuses on special issues, and has not yet addressed issues like manageability, versatility for applications, and streamlining of architectures. The current extension of the XQuery W3C standard for text handling, on the other hand, only considers simple text search functionality without flexible scoring [Ry03, ABS04].

Our design considerations for an algebraic DB&IR interface have been implicitly influenced by earlier work on query algebras for lists [MV93, SLR96, Ra98, Sa01, SJS01], extensible systems with ADTs and user-defined aggregation functions [SLR97, WZ00], and uncertain data with probabilistic reasoning [BGP92, HGS03]. None of this prior work, however, is directly concerned with integrating DB and IR technologies.

2. Application Requirements

2.1 Applications

Potential applications for an integrated DB&IR system are numerous: searching product catalogs for houses, cars, electronics, vacation places, etc.; customer support and customer relationship management (CRM); news archives for journalists and press agencies; personal information management (PIM) for email and file folders, annotated photo collections, etc.; world-wide health care for monitoring and notification of epidemics; bulletin boards for social communities such as ethnic minorities; different flavours of Internet-scale search engines for Deep-Web sources, intranet data, and peer-to-peer search; etc. We discuss three of these example applications, and subsequently derive requirements for a generic platform and an appropriate API on top of which these applications could be built with modest effort.

Customer support. Consider the customer support system of a large software company, electronics manufacturer, or Internet service provider. The system receives help requests and complaints via email or a call-center, such as, e.g., “My notebook, which is model … configured with …, has a problem with the driver of its WaveLAN card. I already tried the following fixes … but only received error messages …” Some of this data can easily be cast into structured fields such as *NotebookModel*, but there remains a good fraction that is free text. Of this text some part may be converted into a semistructured format using natural language processing (NLP) technology; e.g., named entities such as (multi-word and possibly misspelled) product names or temporal adverbial phrases referring to earlier customer requests may be recognized. After capturing the user’s initial request, the system aims to match the new case against previous cases, using similarity measures in terms of customer profiles, the structured part of the request, and the text parts. In addition, the request may be classified into a hierarchy of problem categories, e.g., category /notebooks/drivers/network. When a request cannot be immediately answered, a workflow is established that keeps track of subsequent replies and follow-ups. Similarity search may refer to individual requests or to entire workflows.

News archive. Large press agencies and newspaper publishers produce and receive a huge amount of news and complementary material such as photos every day. A first challenge lies in managing the redundancy in this data, as many articles are just copied and slightly modified. Photos are annotated, at least with respect to time and location (e.g., provided by GPS), and additional annotations may be available in the form of recorded speech. Speech processing technology may transcribe a good fraction of these annotations into text form, and, again, NLP techniques may yield additional structuring and tagging. Of course, news items also carry explicit structured data such as details about their origins (e.g., front page news). This data is used by professional journalists, who are very knowledgeable in their specific fields (unlike, e.g., the typical user for Internet search). Thus, journalists make use of a full repertoire of multilingual search operations, including relevance feedback, rich thesauri, and ontologies. Ontologies are important, for example, in conjunction with spatiotemporal search conditions such as “summer 1998 in Texas” and a relevant news article says “August 1, 1998 in Austin”.

P2P Web search. Consider a peer-to-peer search environment where each peer is a full-fledged search engine, and all peers are autonomous in terms of what data they crawl, index, and score or rank. When a peer seeks information, the request is first evaluated on the peer’s local index, but when the result is insufficient in terms of precision or recall the peer may forward the query to a selected subset of “nearby” peers. This query routing should take into account similarities between peers in terms of their user interests, usage patterns, index contents, etc. Peers can have different search engines but they should all obey some common API. Obviously, peers are interested in exchanging statistical information about their local data, e.g., to estimate the inverse document frequency of words (a standard IR measure) or to compare their index list sizes. Each peer could be programmed and configured in a personalized manner. For example, scores and ranks returned by remote peers could be normalized relative to the peer’s local results and its own scoring function by identifying data objects in the intersection of remote and local results.

2.2 Requirements

With these application scenarios in mind, we now put together a list of requirements for a core DB&IR system that can support development of multiple applications.

1) Flexible scoring and ranking: At the heart of a truly versatile DB&IR system is customizable scoring and ranking. Given the wide spectrum of target applications, it is unlikely that a universal best-compromise solution exists. For example, while Page-Rank-style authority measures are a great asset for Web search, they may be meaningless in an intranet setting where authorship and cross-references are tightly controlled; and a journalist working with a news archive every day may want the system to automatically learn scoring weights according to her personal preferences and relevance feedback. At the API level, explicit control over scoring and score aggregation is essential, despite the widespread belief that only ordinal ranks matter; sophisticated applications such as meta-search engines need to distinguish rankings with all scores close to each other from rankings that have wide gaps in terms of scores. Also, some applications may wish to produce variable-length result lists by thresholding on absolute scores rather than presenting the top k with a fixed k, if some of the top k results are only marginally relevant.

2) Optimizability: Although the emphasis of this discussion is on richer functionality and search result quality, performance is a critical issue for the large-scale applications that we have in mind. Queries in a DB&IR system should be amenable to query optimization that takes workload and data characteristics into account. In particular, scoring and ranking operations need to be viewed in the context of query execution plans, and the optimizer should have leeway for carefully placing these operations. Moreover, it is crucial that, in situations where just a few top-ranked query results are sufficient, the query processor should be able to terminate the computation early without having to enumerate all results. Obviously, this is a highly nontrivial requirement that may not always be achievable, but a good system should at least seize the opportunities that arise in specific cases.

3) Metadata and ontologies: In addition to metadata that describes data sources, additional metadata may be required for building a cognitive model of the user’s information demands, in the form of thesauri, lexicons, ontologies, or at least—in data-warehousing terms—simple dimension hierarchies. A viable long-term solution should also capture uncertainty inherent in such metadata

We believe that the above three requirements, in particular the first, are the most important ones, and their coexistence poses:

Challenge 2: How do we cope with the tension between the need for flexible scoring and ranking and the desire for optimizability of query execution plans?

In addition, a broader view of the topic leads to further requirements that we mention, but will not discuss further in this paper:

4) Data model: Given that words in a text form a sequence, and that ranking of a document collection inherently imposes an order across the documents, we must consider a data model with ordered lists Since documents are often hierarchically structured (e.g., into paragraphs), this naturally calls for an XML-like data model. Unfortunately, the XML standards suite has become overloaded with features and complexity and is thus inappropriate for a manageable API. Inventing a simple core XML model would be a very laudable research task, but "fixing" the entire X suite (XSchema, XQuery, etc.) is a huge order. In this paper we limit ourselves to simple relations and lists of tuples as conceptual data.

5) Data preparation: Raw data like text, speech, or photos often needs to be pre-processed to generate richer annotations, eliminate "fuzzy duplicates", and generally make the data more amenable to effective and efficient querying. Such data preparation could benefit from machine learning techniques and leverage ontological background knowledge and lexicons for named entity recognition, auto-correction of misspellings, etc. A DB&IR system may by itself not include these kinds of techniques, but its API should support easy-to-program interaction with tools and libraries for machine learning and NLP.

6) Personalization: Users have individual preferences. Therefore, flexible mechanisms (e.g., user-specific weights in scoring functions) need to be part of the DB&IR system itself, whereas intelligent strategies for personalization (e.g., learning weights) would naturally belong to the application software.

7) Usage patterns: Considering query logs, click streams, and other statistics from an entire user community may be an invaluable asset for improving the system's search result quality in general. Usage patterns are also a key to personalization strategies. IR research has been much more aware of these user aspects than DB research (see, e.g., [BR99, BFS03, Cha02] and references given there); for integrated DB&IR applications the challenge is widely open.

3. System Architecture

In this section we discuss various design alternatives for the overall architecture of a DB&IR system. As described in the previous section, we are primarily interested in advanced DB&IR applications that need to combine text and structured data, and in scoring and ranking operations that can be flexibly combined and effectively optimized. We do not consider Web search engines as a possible starting point because, while they are intriguing from a software architecture and API viewpoint, they do not provide powerful query primitives such as joins and grouping, let alone in conjunction with scoring, ranking, and cost-based optimization.

We see four major alternatives:

(a) On-top-of-SQL: The IR part of the functionality is layered on top of a full-fledged SQL engine.

(b) Middleware: The combined DB&IR functionality is provided in a middleware layer on top of both an SQL engine and a full-fledged IR system.

(c) IR-via-ADTs: The IR functionality is integrated into an SQL engine using a *separate engine* that is invoked from the relational database via mechanisms such as user-defined functions or ADTs. This is the "ugly" alternative referred to in Section 1.

(d) RISC: Querying capabilities for IR are layered on top of a relational *storage* engine. The latter would be a core system, comparable to, say System R's RSS, Exodus+Shore, or, perhaps, Berkeley DB, with restricted "RISC-style" functionality and a simple API, similar to what we advocated as a base layer in [CW00]. In particular, the storage engine should include self-tuning B-tree-family indexing, full-fledged concurrency control and recovery, and single-table queries with a simple optimizer. The DB&IR system would form an additional layer on top of the storage engine, and application classes would be built on top of the DB&IR API.

We believe that the RISC approach offers the least of the evils. Compared to the other approaches, we find the RISC alternative attractive along two related dimensions: *customizability and programming complexity*, and *efficiency*. Roughly, the first dimension addresses requirements 1) and (especially) 2), and the second addresses requirement 3).

The first three alternatives rely heavily on a full-blown SQL engine, and we believe that this characteristic makes them uninteresting,[†] and in the rest of this section we explain why.

In the ***on-top-of-SQL*** approach, it is difficult to customize the DB&IR functionality in an efficient way. Most notably, if the DB&IR component wants to compute a ranked result list based on its own scoring function, it is forced to extract the complete result from the SQL engine, then score it and sort it, and finally return only a short top-k prefix to the application. If flexible scoring were integrated into the engine, it could employ index pruning techniques to avoid computing the full result set before scoring. Of course, the DB&IR layer could emulate a threshold algorithm on top of the SQL engine, but given that the engine's internal indexes are not visible to the programmer, this would amount to sorted accesses on full

---

[†] As a side remark, note that we do not consider layering DB&IR on top of an XQuery engine, for we believe that this would be even worse than Option (a) in terms of API complexity for the application developer.

tables rather than indexes, and random accesses via primary keys rather than RIDs. It was exactly this kind of indirect and inefficient access to inverted index lists that led Inktomi, a successful Web search engine, to build its own storage system rather than using a SQL engine.

The ***middleware*** approach, albeit seemingly very natural, incurs the complexity of both an SQL engine and a full-fledged IR system. Programmers must cope with two APIs, which may differ radically in terms of their programming philosophies. And, of course, all the performance drawbacks that result from indexes not being accessible by the application code hold in this architecture, too.

The ***IR-via-ADTs*** approach solves the customizability problem "in principle". Well-designed ADTs that build on a core IR engine can potentially allow the programmer to hook application-specific scoring and ranking functions directly into the query processing engine. However, query optimization in the presence of user-defined functions, including row-set aggregation functions, is very difficult. But even from a programmer viewpoint, the IR-via-ADTs architecture is far from ideal: it still has the full complexity of SQL, and performance-conscious coding of ADT functions. In most commercial relational systems, only the core IR functionality (and no additional support for flexible ranking) has been made available through this approach by using a separate IR engine. The IR functionality is programmatically exposed via text as a datatype. Extending this approach to support flexible user-definable ranking functions is not easy. It is also conceivable to extend the SQL engine itself to support IR functionality, e.g., by adding new access methods in the core engine. We do not favor this approach because SQL systems are already complex, and such an extension will lead to significant additional complexity while offering limited flexibility.

In summary, we argue that a storage-level core system with RISC-style functionality is the right architectural approach to explore. Despite its attractiveness, the RISC approach raises some challenges as well. It is no accident that no relational vendor uses its native storage engine for its text ADT. Trying to use the vanilla B+ tree implementation and a traditional relational storage layer without modification can result in very poor performance characteristics. In fact, the challenges may lead us to architecture such as in [Ka03] where interaction between the query (DB&IR API) and the storage layer was different from that in a traditional relational system. Thus, this conclusion leaves us with the challenge:

Challenge 3: Design and implement a "core" storage-level DB&IR platform, with an external API for the query layer, and a carefully designed sharing of responsibilities between the DB&IR layer and the storage engine.

4. Towards an Algebraic DB&IR API

4.1 Framework and Design Rationale

In this section we present and discuss three proposals for a DB&IR core algebra that would enable customizable scoring and ranking, which we regard as *the* key requirement among those identified in Section 2. We cast our considerations into an abstract algebraic framework that we call SALT for "Scoring Algebra for Lists and Text".[‡] As discussed in Section 3, we assume a RISC architecture for an integrated DB&IR system. As for the other requirements, such as interoperability with ontologies, capture and exploitation of usage patterns, etc., we believe that these are partly supported already by using an engine with basic relational database capabilities, and can be enhanced on top of the DB&IR system. For example, we can store thesauri, ontologies, corpus statistics, query logs, click streams, etc. in relational tables and feed them into queries and scoring functions *within* the DB&IR system. We believe that richer NLP and learning techniques can be implemented on top without significant loss of performance, whereas not including customizable scoring in the DB&IR kernel would result in major inefficiencies, as discussed in Section 3.

Conceptually, we view the DB&IR API as an extended form of relational-style algebra, analogous to a storage system providing C++/Java/C# methods for (table or index) scan, selection, projection, etc. Our criteria for a good algebraic interface are: 1) ease of use, which includes concise notation/coding, 2) easy composability with other methods within applications on top of DB&IR, 3) efficiency and optimizability, i.e., the capability to rewrite expressions of the extended algebra and map them onto efficient evaluation plans, and 4) minimal invasiveness with regard to current storage systems. The last point is actually a breach with the clean-sheet-design philosophy of this paper, but industrial experience tells us that designs have a better chance of making an impact if they can be easily combined with existing software and thus preserve earlier investments.

The SALT algebras operate on lists of tuples for two main reasons. First, as discussed in Section 2, lists are more appropriate than sets for modelling the sequencing in text documents.[§] Second, and equally important, as many queries are interested only in the top k elements of a ranked result list, the query processor should be able to stop computations early, rather than first computing entire results and then scoring and sorting them.

So the basic framework should have list-oriented versions of selection σ, join $|\times|$, union $\cup$, difference $-$, and an

---

[‡] Salt was used as a currency before coins became more popular, so it is a traditional means of scoring. Also, the name reminds us of Gerald Salton, the IR pioneer.

[§] It would be tempting to allow nesting of lists, leading to an XML-style data model, but we do not fully understand the ramifications and would like to keep our proposal simple.

additional sort operator to produce lists with specific orders as well as some auxiliary operators like prefix. Note that *order-preservation* is a key issue to be defined for list versions of the above operators. For binary operations such as join we have to distinguish left-order-preservation and right-order-preservation; in special cases the order of both inputs may be preserved. Further, all operators may have both an order-preserving style and a style with arbitrarily (un)ordered output. This allows for alternative implementations with different run-time costs; for example, a hash join would implement a non-order-preserving equijoin whereas index nested-loop join would provide a left-order-preserving equijoin.

We consider three alternative SALT algebras: 1) adding *top-k* and *at-least-k operators*, 2) an extended relational algebra with *scoring/ranking modalities* for each operator, 3) an extended algebra for lists with a single extra *operator* Σ for customizable scoring and ranking.

Note that all major vendors of SQL engines provide a form of top-k syntax for queries with sorted output, e.g., in the form "… order by 2*R.A + 3*S.B stop after 10". This, in combination with user-defined functions, e.g., for specifying "… order by myfunction(R.A, S.B) …", seems to be fully sufficient. Indeed, this approach may be expressive enough, but the crux is in the difficulty of optimizing such queries using the vanilla relational algebraic infrastructure. The point of our discussion is exactly how to package flexible scoring in a way that exposes opportunities to handle top-k queries more efficiently.

4.2 Top-k and At-least-k Operators

The top-k operator proposed by [CK97, CK98] can be generalized to specify how many answers a user wants, but in the context of text, the ranking function can be quite complex. Given a query with a final top-k clause based on an application-specific scoring function, we could derive additional filters for the attributes used in the scoring function. This amounts to mapping top-k queries into multidimensional selections with **speculative filters** [BCG02, CG96, CK98, DR99]. For example, the system could automatically rewrite a top-k query about recent sports news in Paris into

σ[Category="sport" ∧ Date ≥ 15-March-2004
∧ distance(Location, "Paris") ≤ 200km] (News)

where Category, Date, and Location are attributes of a News table, and *distance* is a user-defined function.

The difficulties with this approach are threefold. First, while numerical attributes are straightforward to handle, it is unclear how to deal with categorical attributes and similarity scores that are highly application-dependent. For Location we resort to a distance function above, but the application may prefer scores derived from an ontological generalization hierarchy so that, for example, Location "Isle de France" is considered a good match to the query condition. Second, it is extremely difficult to derive suitable filter predicates from an arbitrary score aggregation function, which may be more complex than merely a weighted sum. Selectivity estimation would have to consider correlations among the attributes in the filter but also with non-relaxed attributes (e.g., between Category and Date) and text attributes (e.g., the occurrence of words "soccer", "cup", "champion" in a Headline attribute). If selectivities are overestimated, the query produces too few results and has to be restarted. Third, it is unclear how speculative filters should be generated for more complex query expression that involve unions, differences, outer joins, grouping, etc.

Solutions to these problems are not out of the question, but pose major research challenges, such as:

Challenge 4: How can we accurately predict the effect of speculative filters on result sizes for arbitrarily complex queries and application-specific scoring functions?

The top-k clause allows the programmer to request the most interesting few results, but what if there are *fewer* than the desired number of answers? We can extend every operator of a core relational algebra like selection σ, union ∪, intersection ∩, difference −, equijoin |×|, etc. to become a **stretchable operator** so that it returns at least k (or, ideally, exactly k) result tuples, where k can be set by the programmer.

For example, we would write σ~[k,F](R) for a stretchable top-k selection on input R with filter formula F, ∩~[k](R,S) for a stretchable intersection, and so on. The semantics is that such an operator returns the best k *approximate* results if the corresponding exact-match operator returns fewer than k results. So the stretchable operators serve to guard against empty or almost empty results by relaxing filter conditions, join conditions, intersection semantics, etc. Every stretchable operator could be parameterized with a similarity measure on the corresponding domains, for example, on a Genre domain in the case of query σ~[k,Genre="Science Fiction"] (Movies) or on the Cartesian product of the attribute domains for two schema-compatible relations in the case of query ∩~[k] (Books1, Books2) with a similarity function that takes into account misspellings, synonyms, etc.

This idea is appealing at first glance, but becomes fairly complex upon deeper thought. First, each operator would have to provide scored result tuples, as the application programmer may want to combine scores from different expressions, and it is not clear how to make this composable. Likewise, it is unclear how the programmer should choose the parameter k for different operators in a composite expression. Ideally, the programmer would like to specify k only for the outermost operator and leave the optimizer to "fill in" the scoring parameters for all the inner operators. For example, a query on a movie database that includes IR search conditions on both structured attributes like year and text data like customer reviews could look as follows:

σ~[k,True] (|×|~[k3, Movies.Title=Reviews.Title]
(σ~[k1,Movies.Year=1999](Movies),

$$\sigma\text{~}[k2,\text{Reviews.Text like “…”}](\text{Reviews})\ )\ )$$

This requires fine-tuning the k1, k2, and k3 parameters in order to ensure that the outermost selection σ~[k,True] can return k results. Note that we need this outermost selection, with a trivial filter condition, for the final combination of scores. Further, note that the equijoin needs to be stretchable, too, in the sense that it considers tuples as approximately matching even if their titles differ slightly.

The proper choice of the target size for intermediate results is intrinsically related to the difficult issue of accurately estimating query result cardinalities and value distributions (e.g., using advanced histograms or other forms of data synopses). Despite the rich body of research on approximate query processing there is no satisfactory answer yet to these kinds of estimation problems, and the additional aspect of flexible scoring functions makes the problem even harder.

In summary, the stretchable operator approach leaves us with the following major research issues:

Challenge 5: How are result-size targets propagated between operators in a complex query execution plan? If some top-level operator should produce k results, how can we estimate the desired number of results k' > k that the various subordinate operators need to deliver?

4.3 Operators with Scoring Modalities

The scoring function can be viewed as a *modality* that can be attached to any relational operator.[**] If the scoring depends only on the output of the regular operator, e.g., scoring the result of a selection, then it suffices to consider scoring as just another operator. However, if scores may depend on input as well as output tuples of a regular operator, then viewing scoring as a modality of the operator seems a natural approach.

For example, consider a union operator which we want to extend with scoring as follows: tuples that appear in both input relations obtain score 2 and all other result tuples obtain score 1. This scheme could be generalized in many ways. For example, both inputs could already carry scores, and a result tuple that appears in both inputs is assigned the average of its input scores, whereas a result tuple that appears in only one input is assigned the score from that input minus some constant penalty c. The point is that it is impossible to compute this kind of scoring *after* we have computed the result of the union (unless we access additional data like the intersection result). Rather, the scoring must be tied in with the union operator itself and see the input tuples of ∪, too. A conceivable solution is to have a scoring function invoked on each of the two inputs and an additional score aggregation function on the output tuples that are produced. The latter would simply sum up the scores for those tuples that appear in both inputs. In ad-hoc notation this modality-based scoring would look as follows:

$$\begin{array}{l} \cup[\text{score}(t) = \text{if } t \in R\cup S \text{ then scoreR}(t) + \text{scoreS}(t) \\ \qquad\qquad \text{else if } t \in R \text{ then scoreR}(t) - c \\ \qquad\qquad \text{else if } t \in S \text{ then scoreS}(t) - c\ ] \\ \quad (\ R[\text{scoreR}(t) = \dots\ ],\ S[\text{scoreS}(t) = \dots\ ]\ ) \end{array}$$

The function definitions in squared brackets are the modalities of the operators; for the two union operands we could also have algebraic expressions rather than base tables, e.g., σ[F][scoreσR(t)=…](R[scoreR=…]). Note that the key point of this approach is that we can apply these considerations to each and every basic operator that our regular, exact-match, algebra provides. Further, we can customize the scoring modalities to each operator, even every invocation of an operator, in an application program. A scoring function may even invoke another database query to access score-relevant auxiliary information, such as idf values or synonyms for query terms.

The elegance of the approach lies in the fact that scoring is added to the original application queries in a perfectly orthogonal manner; dropping the modalities results in an exact-match query without leaving traces such as auxiliary operations that were introduced solely for the purpose of similarity search. Thus, an application could easily switch between exact-match evaluation mode and similarity-search mode, by recompiling queries. Making this idea practical requires us to provide a formalization of scoring modalities for relational operators that is at once expressive and intuitive, leading to our next challenge:

Challenge 6: Develop a full-fledged formal semantics for the scoring modality model. Develop typical use-case templates to guide programmers towards readable and correct usage.

4.4 The Σ Operator

In this section, we present the third approach towards a SALT algebra: the Σ operator, a generalization of relational selection σ. The Σ operator takes a single input list and produces a single output list. The output may be shorter than the input and its schema is extended by a score attribute which is uniquely named. Σ has four programmable and thus customizable arguments:

1. A *set* α *of simple aggregation functions*, where each function is restricted to be an *accumulator* whose value can be computed by linear recursion over prefixes of the input list.[††] Thus, every accumulator function has time complexity O(n) where n is the size of the input list, and its space complexity is required to be O(1); that is, it can use only constant working memory. (We will explain below the rationale for these restrictions.)

[**] The use of the term “modality” is analogous to modal logics, e.g., CTL, where modalities are associated with quantifiers.

[††] We note that LDL++ [ZAO93, WZ00] proposed a recursive approach to new user-defined aggregate operations, similar to our proposal for user-defined scoring operators. However, they produce one aggregate result per input collection, whereas we consider prefixes of lists. Also, they did not consider algebraic compositionality, or mechanisms to facilitate early termination.

2. A *scoring function ρ* from $\mathrm{dom}(R)\times\mathrm{out}(\alpha)$ to real numbers (or non-negative real numbers or the interval [0,1] if normalization is desired). Here $\mathrm{out}(\alpha)$ denotes the Cartesian product of the result domains of the accumulators in α. The result of ρ depends only on the values in a given tuple and the values of the accumulators when seeing this tuple. Note that we do not *a priori* rule out negative scores, for this might be useful to express anti-preferences.
3. A *filter condition F*, which can be a Boolean combination of elementary comparisons of the form *attribute θ constant* or *attribute θ attribute*, the same kind of formulas that we are allowed to use in selections. F refers to a single tuple only, that is, it must be completely evaluable without accessing other tuples. However, a point specific to Σ is that F may refer to the values of the accumulators in α.
4. A *stopping condition T*, which is of the same format and complexity as F.

When applied to input list R, we may write $\Sigma[\alpha;\rho;F;T](R)$. For example, a ranked search for movies that appeared in or around a given year and obtained reviews with certain indicative terms could be phrased as follows:

```
sort[k, Score, desc] (
   Σ[α: min := min{Score of best k tuples so far};
          threshold := best possible Score of
                             candidates not yet seen at all;
          count := length of current prefix;
      ρ(t) := sum(MoviesScore, ReviewsScore);
      F: Score > min ∨ count < k;
      T: (min ≥ threshold  ∧ count ≥ k]  (
         merge (sort[MoviesScore] (Σ[...](Movies)),
                   sort[ReviewsScore] (Σ[...](Reviews))
                  ) ) )
```

Here an appropriate threshold is computed for early stopping, such as the threshold of the TA method. This illustrates the versatility of the accumulator and stopping-condition concepts. Note that Σ is applied in a nested way, computing the sub-scores MoviesScore and ReviewsScore on the two lists that form the merged input stream to the outer Σ. This illustrates the easy composability of the Σ operator. All operators consume lists in a pipelined manner. The outermost sort operator is needed because the filter F is passed by top-k candidates that may later be superseded by better results. Note, however, that this sorting can be implemented by a bounded priority queue, so it is very space-efficient and does not block the pipeline.

All four building blocks α, ρ, F, and T are *applied to each prefix* of the input list R. We describe the semantics of Σ in the following procedural manner:

Step 1: Accumulator computation. For prefix p of R with t being the currently seen tuple (i.e., p's suffix of length 1), the accumulators in α are computed. Because of their linear-recursion property, this can be done with the previous accumulator values and t as the only inputs. One possible purpose of the accumulators is to track thresholding values such as higher and lower bounds for scores of interesting tuples, e.g., the current top-k tuples. Note that it is crucial to limit the space complexity of α to $O(1)$, with k for top-k being viewed as a constant, but the length of the current prefix would violate the $O(1)$ restriction. Otherwise, scoring would be a treacherous backdoor for inefficient queries, an aspect that would be awfully hard for the query optimizer to deal with. In fact, this is exactly the kind of trapdoor that we see in the IR-via-ADTs approach of Section 3 and have criticized there.

Step 2: Scoring. Next we compute the actual score of the current tuple t, based on t's attribute values and the values of the accumulators. The latter are simply treated as if they were additional virtual attributes of t. Note that the computation of ρ cannot use any global information that is not carried in t or the accumulators. This rules out, for example, accessing some global statistics for obtaining idf values and factoring them into a list of tf-scored documents. To achieve this effect, the programmer would first have to construct input tuples to Σ that already carry the idf values with them (using the SALT algebra). The rationale for this restriction is to make expensive operations (e.g., joins in this case) explicit to the programmer and explicitly known to the query optimizer.

Step 3: Result tuple preparation. The result of the scoring function ρ is made available in the output tuples of Σ in the form of an additional attribute "Score" whose actual name can either be chosen by the programmer by an "As <Name>" clause or is automatically generated. We assume that the names of such virtual attributes produced during query processing are unique within the given query. Analogously, results of accumulator functions in α could be made available in output tuples, too.

Step 4: Filter test. For each prefix the current tuple is tested against the filter condition F. It is output by the Σ operator only if it passes the test; otherwise it is discarded. Note that the filter test is applied after the result tuple is prepared so that it can more easily refer to accumulator and scoring values by named attributes.

Step 5: Stopping test. For each prefix, the stopping test T is executed. Like the filter test, it can refer to the values of the current tuple and the named results of accumulators and scoring. If the stopping test yields true, the entire operator execution is terminated. The role of T is obviously to provide the programmer with flexible ways of threshold-driven (or other, provably safe or heuristic, approaches to) early termination. After all, top-k similarity queries are heuristic by their very nature: users rarely look at all top k results in detail but would merely like to spot one or two good results among the top k. This may justify all kinds of application-specific pruning tricks for efficiency. Recall that the SALT framework is designed to accommodate customizable pruning, but by itself provides only mechanisms to this end and no pruning strategies.

The mechanisms we described for the Σ operator resemble the rank( ) function of SQL:1999 in combination with user-defined aggregation functions for scoring. Indeed, all user-defined aggegation in SQL follows the same initialize-accumulate-terminate steps that Σ uses. However, Σ is more powerful as it supports filter conditions that can depend on accumulator values and, most importantly, allows early termination by the programmable stopping test.

We believe that we have identified a valuable set of features and packaged them into the relatively lightweight Σ operator, which offers flexibility while allowing for integration into a relational engine with modest effort. However, we invite the reader to disagree and tackle:

Challenge 7: Apply Occam's razor to the Σ operator and identify core functionality such that we ideally achieve 80 percent of the benefits with 20 percent of the current complexity. Or design an alternative to Σ!

4.5 Discussion

It is not easy to compare our proposals for a SALT algebra; they have very different appeals and pitfalls, and above all, we have only a preliminary and fairly superficial understanding of their properties. Table 1 depicts our immature and vague assessments in a simplified manner, using the qualitative ratings +, 0, - from good to bad. We break down the ease-of-use criterion into two separate aspects: how easy it is for the programmer to write reasonable code for simple tasks using the SALT algebra, and how easy it is to fully understand the SALT concepts and program difficult tasks with high confidence in the correctness of the solution. In a similar vein, we decompose the system complexity and performance issue into two aspects: how easy it is to implement the SALT algebra in a storage-level system, and how easy it is to reflect its impact in the query optimizer.

	Programming	Understanding	Flexibility	Implementation	Optimization
Top-k & at-least-k operators	+	0	–	+	–
Modalities	0	–	+	0	–
Σ operator	0	0	+	+	0

Table 1: Comparison of SALT algebras

Speculative filters for top-k operators are easy to use, ideally transparent to the programmer if automatically generated, but query results may not be easily explainable to the programmer and it is unclear how to incorporate arbitrary scoring functions. Moreover, selectivity estimation is still a big open issue for speculative filters. Stretchable at-least-k operators are a reasonably straightforward abstraction for simple tasks. However, their limits in terms of expressiveness are unclear, and the query optimizer faces very difficult result-size estimation problems. Modalities for every operator seem to have the highest conceptual complexity; writing programs may still be manageable by many but fully understanding all potential intricacies seems challenging. How to efficiently implement and optimize modalities is a widely open issue. The Σ operator is in some sense a light-weight, more easily digestible variant of the modalities approach, and we fleshed it out in more detail than the other proposals. It compares favorably to modalities in terms of programming complexity, and we believe that it is better suited for DB-style query optimization than the other approaches.

5. Optimization Issues

One of the cornerstones of efficient query processing is *pipelining* between operators, to avoid materializing big intermediate results. This is of utmost importance for ranking queries where we only want the k best results, and would like to stop all query processing as early as possible. Fortunately, all three of our proposals for SALT algebras are well behaved with regard to pipelining; in fact, they were designed with this in mind. Thus, when a SALT expression requires sorted input to a stretchable operator, an operator with scoring modality, or the Σ operator (even if this input has to be joined with big tables), the scoring part of the expression can consume the input incrementally and stop as soon as it is sure it has found the top k results. With sorting as a subordinate operator, this is, of course, only feasible if the sorted stream can be produced by an index or table scan without explicit sorting. The speculative filter approach behaves just like any other selection and so is naturally pipeline-enabled.

Orthogonal to pipelining is the ability of a query optimizer to push selective operators toward the leaves of an operator tree and, more generally, reorder expensive operators so as to minimize total execution costs. This is done through *algebraic equivalences* that serve as rewrite rules. For example, when a scored result is order-correlated to the input sorted by certain attributes, and this sort order can be produced by an inexpensive index scan, then we would like to commute sorting and scoring in an expression such as sort[…] (Σ[…] (…)). Or we would like to commute selection and scoring in σ~[k, F] (σ[f(…) As Score] (…)), with the stretchable operator approach, if the filter predicate F is highly selective.

Unfortunately, there are few algebraic equivalences for our SALT algebras. For example, σ and Σ are not commutative, unless the score function and the stopping condition were restricted to allow only trivial and useless cases. Likewise, none of the scoring or stretchable operators commute with sorting. Does this mean that our various SALT algebras are not well designed, and that we should look for better alternative algebras? Perhaps, but we believe, rather, that it reveals an inherent difficulty in the nature of scoring and ranking operators in general. In formalizing algebraic properties, we had to deal with lists and bags and order-preserving or non-order-preserving relational operators, and we realized that the literature is very scarce on truly rigorous results for ordered bags (a notable exception being [SJS01]). This leads us to:

Challenge 8: Develop a comprehensive set of algebraic equivalences for a SALT-like DB&IR algebra, in particular, equivalences that facilitate early termination. If the algebra exhibits only weak properties in terms of rewritability, develop useful sub-algebras and specialized, restricted variants of operators that are sufficiently expressive and have better opportunities for rewriting.

An alternative is to relax the operators' semantics of being exact computations. Instead, we could consider them as merely approximative, in the sense of contributing to a correct *top-k* query result only *with high probability*. We believe that most target applications of a DB&IR system would justify such a relaxation, but it is important to be precise about the nature of such approximability.

Challenge 9: Develop a notion of "approximative equivalences" and "approximative orders" that hold with high probability (and controllable error bounds). Study how this would affect the rewriting capabilities of a DB&IR query optimizer.

The last challenge may build on earlier work about data synopses and approximate query processing (e.g., [AGP99, Ch01, CDN01, HH01]), but it aims at a wider and more demanding target and it may perhaps be better to pursue radical departures from these earlier approaches.

6. Conclusion

This paper is a high-level attempt to lay out a research agenda for integrating DB and IR technologies, with particular emphasis on the tradeoffs between customizable scoring and optimizability. We believe we have pointed out a number of interesting research opportunities, even though—or perhaps *because*—our concrete proposals (in particular, the various algebras) are clearly far from conclusive. Some of the challenges that we pose throughout the paper require major community efforts, but some may be tackled at the level of individual doctoral theses. We think that Challenges 4 through 8 are in the latter category, whereas the other challenges are at a more strategic level. In conclusion we add a final community-scale issue:

Challenge 10: How can we foster better interaction between, and eventually integration of, the sociologically separated DB and IR research communities?

The challenges raised here are central to understanding whether DB and IR are destined to forever be on two separate islands, or whether they share far more synergy than is apparent today.

References

[AGP99] S. Acharya, P.B. Gibbons, V. Poosala: Aqua: A Fast Decision Support Systems Using Approximate Query Answers. VLDB 1999

[As02] B. Aditya, G. Bhalotia, S. Chakrabarti, R. Desai, A. Hulgeri, C. Nakhe, Parag, S. Sudarshan. Browsing and Keyword Search in Relational Databases. VLDB 2002

[ACD02] S. Agrawal, S. Chaudhuri, G. Das: DBXplorer, A System for Keyword-Based Search over Relational Databases, ICDE 2002

[Ag03] S. Agrawal, S. Chaudhuri, G. Das, A. Gionis: Automated Ranking of Database Query Results. CIDR 2003

[AYJ03] S. Al-Khalifa, C. Yu, H.V. Jagadish: Querying Structured Text in an XML Database. SIGMOD 2003

[ACS02] S. Amer-Yahia, S. Cho, D. Srivastava: Tree Pattern Relaxation. EDBT 2002

[Am03] S. Amer-Yahia, M.F. Fernandez, D. Srivastava, Y. Xu: Phrase Matching in XML. VLDB 2003

[ABS04] S.Amer-Yahia, C. Botev, J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. WWW 2004

[BR99] R. Baeza-Yates and B. Ribeiro-Neto. Modern Information Retrieval. Addison-Wesley, 1999

[BFS03] P. Baldi, P. Frasconi, P. Smyth: Modeling the Internet and the Web, Wiley&Sons, 2003

[Ba03] A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, T. Wang: A System for Keyword Proximity Search on XML Databases. VLDB 2003

[BHP04] A. Balmin, V. Hristides, Y. Papakonstantinou: ObjectRank: Authority-Based Keyword Search in Databases, VLDB 2004

[BGP92] D. Barbará, H. Garcia-Molina, D. Porter: The Management of Probabilistic Data. IEEE TKDE 4 (5), 1992

[BP98] S. Brin, L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. WWW 1998

[BCG02] N. Bruno, S. Chaudhuri, L. Gravano: Top-k selection queries over relational databases: Mapping strategies and performance evaluation. ACM TODS 27 (2), 2002

[BGM02] N. Bruno, L. Gravano, A. Marian: Evaluating Top-k Queries over Web-Accessible Databases. ICDE 2002

[Ca03] D. Carmel, Y.S. Maarek, M. Mandelbrod, Y. Mass, A. Soffer: Searching XML Documents via XML Fragments, SIGIR 2003

[CH02] K. Chang, S. Hwang: Minimal probing: supporting expensive predicates for top-k queries. SIGMOD 2002

[CK97] M.J. Carey, D. Kossmann: On Saying "Enough Already!" in SQL. SIGMOD 1997

[CK98] M.J. Carey, D. Kossmann: Reducing the Braking Distance of an SQL Query Engine. VLDB 1998

[Ch01] K. Chakrabarti, M.N. Garofalakis, R. Rastogi, K. Shim: Approximate query processing using wavelets. VLDB Journal 10 (2-3), 2001

[Cha02] S. Chakrabarti: Mining the Web: Discovering Knowledge from Hypertext Data, Morgan Kaufmann, 2002

[CDN01] S. Chaudhuri, G. Das, V. Narasayya: Vivek R. Narasayya: A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries. SIGMOD 2001

[CDY95] S. Chaudhuri, U. Dayal, T.W. Yan: Join Queries with External Text Sources: Execution and Optimization Techniques. SIGMOD 1995

[CDH04] S. Chaudhuri, G. Das, V. Hristides, G. Weikum: Probabilistic Ranking of Database Query Results, VLDB 2004

[CG96] S. Chaudhuri, L. Gravano: Optimizing Queries over Multimedia Repositories, SIGMOD 1996

[CW00] S. Chaudhuri, G. Weikum: Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. VLDB 2000

[Ch04] S. Chaudhuri, G. Das, V. Hristides, G. Weikum: Probabilistic Ranking of Database Query Results, VLDB 2004

[DR99] D. Donjerkovic, R. Ramakrishnan: Probabilistic optimization of top N queries. VLDB 1999.

[CK02] T.T. Chinenyanga, N. Kushmerick: An expressive and efficient language for XML information retrieval. JASIST 53 (6), 2002

[Co98] W.W. Cohen: Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. SIGMOD 1998

[Co00] William W. Cohen: Data integration using similarity joins and a word-based information representation language. ACM TOIS 18(3), 2000

[Co03] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv: XSEarch: A Semantic Search Engine for XML. VLDB 2003

[CL02] W.B. Croft and J. Lafferty (eds), Language Models for Information Retrieval, Kluwer Academic Publishers, 2002

[FKS03] R. Fagin, R. Kumar, D. Sivakumar: Efficient similarity search and classification via rank aggregation. SIGMOD 2003

[Fa03] R. Fagin, R. Kumar, K.S. McCurley, J. Novak, D. Sivakumar, J.A. Tomlin, D.P. Williamson: Searching the workplace web. WWW 2003

[FLN03] R. Fagin, A. Lotem, M. Naor: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. 66(4), 2003

[FKM00] D. Florescu, D. Kossmann, I. Manolescu: Integrating keyword search into XML query processing. WWW 2000

[Fu00] N. Fuhr: Probabilistic Datalog: Implementing Logical Information Retrieval for Advanced Applications, JASIS 51(2), 2000

[FG04] N. Fuhr, K. Großjohann, XIRQL: An XML Query Language Based on Information Retrieval Concepts, ACM TOIS

[FGA03] N. Fuhr, N. Gövert, M. Abolhassani: Retrieval Quality vs. Effectiveness of Relevance-Oriented Search in XML Documents, Tech. Report, Univ. Duisburg-Essen, 2003.

[GBS01] T. Grabs, K. Böhm, H.-J. Schek: PowerDB-IR - Information Retrieval on Top of a Database Cluster. CIKM 2001

[GJM97] W.I. Grosky, R. Jain, R. Mehrotra, Handbook of Multimedia Information Management, Prentice Hall, 1997

[Gu03] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram: XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003

[GBK00] U. Güntzer, W.-T. Balke, W. Kießling: Optimizing Multi-Feature Queries for Image Databases. VLDB 2000

[HH01] P.J. Haas, J.M. Hellerstein: Online Query Processing. SIGMOD 2001

[HKP01] V. Hristides, N. Koudas, Y. Papakonstantinou: Prefer: A System for the Efficient Execution of Multi-parametric Ranked Queries. SIGMOD 2001

[HP02] V. Hristides, Y. Papakonstantinou: DISCOVER: Keyword Search in Relational Databases. VLDB 2002

[HGS03] Edward Hung, Lise Getoor, V.S. Subrahmanian: Probabilistic Interval XML. ICDT 2003

[IAE03] I.F. Ilyas, W.G. Aref, A.K.Elmagarmid: Supporting Top-k Join Queries in Relational Databases. VLDB 2003

[Il04] I.F. Ilyas, R. Shah, W.G.Aref, J.S. Vitter, A.K. Elmagarmid: Rank-Aware Query Optimization, SIGMOD 2004

[Ka03] Navin Kabra, Raghu Ramakrishnan, Vuk Ercegovac: The QUIQ Engine: A Hybrid IR DB System. ICDE 2003.

[Ka04] R. Kaushik, R. Krishnamurthy, J. Naughton, R. Ramakrishnan: On the Integration of Structure Indexes and Inverted Lists. SIGMOD 2004

[Ki02] W. Kießling: Foundations of Preferences in Database Systems. VLDB 2002

[LK02] A. Leubner, W. Kießling: Personalized Keyword Search with Partial-Order Preferences. SBBD 2002

[LS03] X. Long, T. Suel: Optimized Query Execution in Large Search Engines with Global Page Ordering. VLDB 2003

[Lo03] Authors: The Lowell Database Research Self Assessment, 2003, http://research.microsoft.com/~gray/lowell/

[MV93] D. Maier, B. Vance: A Call to Order. PODS 1993

[MS99] C.D. Manning, H. Schütze: Foundations of Statistical Natural Language Processing, MIT Press, 1999

[MZ96] A. Moffat, J. Zobel: Self-Indexing Inverted Files for Fast Text Retrieval. ACM TOIS 14(4), 1996

[Na01] A. Natsev, Y.-C. Chang, J.R. Smith, C.-S. Li, J.S. Vitter: Incremental Join Queries on Ranked Inputs. VLDB 2001

[NR99] S. Nepal, M.V. Ramakrishna: Query Processing Issues in Image (Multimedia) Databases. ICDE 1999

[Ra98] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K.S. Beyer, M. Krishnaprasad: SRQL: Sorted Relational Query Language. SSDBM 1998

[Ry03] M. Rys: Full-Text Search with XQuery: A Status Report. H. Blanken et al. (Eds.): Intelligent Search on XML Data, 2003

[Sa01] R. Sadri, C. Zaniolo, A.M. Zarkesh, J. Adibi: Optimization of Sequence Queries in Database Systems. PODS 2001

[SLR96] P. Seshadri, M. Livny, R. Ramakrishnan: Design and Implementation of a Sequence Database System. VLDB 1996

[SLR97] P. Seshadri, M. Livny, R. Ramakrishnan: The Case for Enhanced Abstract Data Types. VLDB 1997

[SM02] T. Schlieder, H. Meuss: Querying and ranking XML documents. JASIST 53(6), 2002

[STW03] R. Schenkel, A. Theobald, G. Weikum: Ontology-Enabled XML Search. In: H. Blanken et al. (Eds.): Intelligent Search on XML Data, 2003

[SJS01] G. Slivinskas, C.S. Jensen, R.T. Snodgrass: A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering. IEEE TKDE 13(1), 2001

[TW00] A. Theobald, G. Weikum: Adding Relevance to XML. WebDB 2000

[TW02] A. Theobald, G. Weikum: The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002

[TWS04] M. Theobald, G. Weikum, R. Schenkel: Top-k Query Evaluation with Probabilistic Guarantees, VLDB 2004

[WZ00] Haixun Wang, Carlo Zaniolo: User Defined Aggregates in Object-Relational Systems. ICDE 2000

[WMB99] I. Witten, A. Moffat, T. Bell: Managing Gigabytes: Compressing and Indexing Documents and Images, 2ed, Morgan-Kaufmann, 1999

[YPM03] C.T. Yu, G. Philip, W. Meng: Distributed Top-N Query Processing with Possibly Uncooperative Local Systems. VLDB 2003

[ZAO93] C. Zaniolo, N. Arni, K. Ong: Negation and Aggregates in Recursive Rules: the LDL++ Approach. DOOD 1993

[ZAR03] P. Zezula, G. Amato, F. Rabitti: Processing XML Queries with Tree Signatures. H. Blanken et al. (Eds.): Intelligent Search on XML Data 2003