Bulletin of the Technical Committee on

# Data Engineering

December 2001 Vol. 24 No. 4

IEEE Computer Society

## Letters

## Special Issue on Text and Databases

## Conference and Journal Notices

## Editorial Board

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

The Data Engineering Bulletin web page is http://www.research.microsoft.com/research/db/debull.

## TC Executive Committee

# Letter from the Editor-in-Chief

## TCDE Election

As you can see from the short letter from the TCDE nominating committee (on page 2), Erich Neuhold has been elected as chair of the IEEE Techincal Committee on Data Engineering. His term will start in January, 2002. Congratulations Erich on your new position! Erich has been serving as TCDE vice chair, and is the current chair of the ICDE Steering Committee.

I also want to take this opportunity to thank Betty Salzberg, our current TCDE chair, whose term ends this month. Betty has done an outstanding job of re-invigorating the technical committee. We have a TCDE web site, we have a regular meeting at our TCDE sponsored conference ICDE. We have also had meetings to discuss issues related to TCDE finances, conference sponsorships, and member services. During Betty's term, two significant decisions were made relative to our publications. First, the Bulletin is now an entirely electronic publication. This is important as it means that our expenses drop dramatically. (The Bulletin was the largest item by far in the TCDE budget.) Second, the TCDE was then able to sponsor the distribution of the SIGMOD Anthology to its members (who were not also SIGMOD members). These were major accomplishments of Betty's leadership over the past four years.

## The Current Issue

The database technical field had its start in business data processing. Databases have, over time, gotten faster and faster at handling larger and larger volumes of business transactions. Along with these improvements in performance metrics, databases have evolved into search engines that are increasing capable of dealing with complex, decision support style queries. This gave rise to the OLAP community, data warehouses, and data mining. Increased functionality has also been an on-going theme in database evolution. And we have seen databases applied to time-series data, geographical data, etc.

One of the truly challenging areas to which database have been applied is the area of text search, and its integration with move conventional structured queries. Information retrieval (IR) is a field that is even older than databases. But IR did not attempt this integration. IR itself is a hard problem. Its integration with database search adds additional complication. However, progress is being made. And database vendors sense that there is a real market for this unified search capability. Hence, the current issue on ”Text and Databases” is very timely.

Luis Gravano, the December issue editor, has himself worked in this area. He has used his knowledge of the area in bringing together a very interesting sample of the work going on in unifying text and database search. I particularly appreciate that Luis has successfully coaxed database vendors to contribute to this issue, thus keeping our field apprised of the commercial state-of-the-art. I want to thank Luis for his very successful effort in bringing us all this report on where we stand with text and database search.

David Lomet
Microsoft Corporation

## New TCDE Chair for 2002-2003

The election for Chair of the IEEE Computer Society Technical Committee on Data Engineering (TCDE) for the period January 2002 to December 2003 has concluded. We are pleased to announce that Professor Erich Neuhold, Darmstadt University of Technology (neuhold@darmstadt.gmd.de) has been elected Chair of the TCDE. Erich's biography and a position statement can be found in the September 2001 issue of the DE Bulletin.

Paul Larson, Masaru Kitsuregawa, Betty Salzberg
Nominating Committee

## Letter from the Special Issue Editor

Text is everywhere: in web pages, in unstructured and semistructured documents, and in structured objects. This issue of the Bulletin addresses a few of the diverse challenges associated with handling text effectively. The first four papers in the issue, by Albert Maier and David Simmen (IBM), James Hamilton and Tapas Nayak (Microsoft), Paul Dixon (Oracle), and Prabhakar Raghavan (Verity), provide a revealing industry perspective on text management. Specifically, these four papers describe how IBM's DB2, Microsoft's SQL Server, Oracle, and Verity's K2 Enterprise handle textual data, focusing particularly on the integration of text and structured data. The next two papers are on approximate string matching, a critical problem when processing text-related queries. For example, textual data may be riddled with typographical errors or present many ways of referring to the same person or organization, in absence of universally adopted conventions or formats. The paper by Gonzalo Navarro et al. is a survey of specialized index structures for approximate string matching. The paper by Panagiotis Ipeirotis et al. discusses how to process approximate string join and selection queries over a standard, unmodified relational database management system. At a higher level of abstraction, the information retrieval field has studied for decades how to answer queries effectively over large collections of (flat) text documents. The paper by Amit Singhal (Google) is a survey of the latest developments in information retrieval. This paper includes a detailed description of a state-of-the-art document scoring scheme, which should help researchers design experimental evaluations in this general topic. Finally, the last paper in the issue, by Sriram Raghavan and Héctor García-Molina (Stanford U.), surveys work on the integration of collections of unstructured text documents and databases of semistructured or structured data.

This issue of the Bulletin touches on just a few of the many topics that are relevant to the management of textual data. I have intentionally shifted the focus of the issue away from web-specific aspects of the problem, which were at the core of the September 2000 issue of the Bulletin on "Next-Generation Web Search." I hope you will find that the eight papers in the current issue of the Bulletin provide a stimulating peek at the research and development that is happening in both the industrial and the academic worlds.

Luis Gravano
Columbia University

# DB2 Optimization in Support of Full Text Search

Albert Maier
IBM Boeblingen Lab
amaier@de.ibm.com

David Simmen
IBM Silicon Valley Lab
simmen@us.ibm.com

## 1 Introduction

Over the last several years, demand for integrating full text and relational search has increased dramatically. For example, a content management system might seek the most relevant articles written during a certain period, where the abstract contains the words 'Bush' in the same sentence as 'recession'. Neither a traditional text information retrieval (IR) system nor a traditional RDBMS could handle this type of query efficiently.

Not surprisingly, federated solutions emerged where an application layer decomposed queries into relational and text search components and joined the results. IBM's Digital Library exploited DB2 and IBM's powerful linguistic text search engine (TSE) in this way. One of the inherent drawbacks of this architecture is the cost of moving partial results into the application layer and joining them there.

DB2's Text Extender (TE) was the first attempt by a major RDBMS vendor to tackle this problem. TE integrates TSE into DB2. It supports creation of text indexes on DB2 tables. These text indexes use keys stored in the table to relate index entries back to individual records. Triggers are used to keep the text and text indexes in sync. TE exploited DB2's SQL extensibility features to effectively push query decomposition and join into the RDBMS. Following is an example of a typical TE query that performs the "recession" search described above:

```
SELECT a.isbn, a.year, b.score
FROM articles a, text_search('articles','abstract','"Bush" in same sentence as "recession"') b
WHERE a.year >= 2000 AND a.isbn = b.key AND b.score >= 0.9
ORDER BY 3
```

The table function *text_search* calls TSE to evaluate the text search condition, returning the keys of all matching articles together with a relevance score. This approach allows text and relational results to be joined using the sophisticated join methods of the RDBMS; however, optimal decomposition of the query, e.g., choice of access order, join methods, etc. requires information regarding the cost and cardinality of the text search be made available to the query optimizer. Note that performance of the query would also benefit if DB2 could make TSE aware that the query was interested in only the most relevant articles having a certain minimum score. So in addition to the cost estimation issues that effected performance, there were lost opportunities to limit cross-source data flow by pushing query processing (e.g., predicate application) to TSE.

These federated query optimization issues have been investigated extensively since TE was released. Of particular relevance here is the Garlic work [1], which described a general framework for performing cost-based

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

optimization of queries across heterogenous data sources. We exploited basic ideas from this work in order to solve many of the TE optimization issues described above. In particular, like Garlic, we did not wire knowledge about the capabilities of the text search engine into the DB2 optimizer. Instead, we added an API that allows the text search engine to exchange costing and planning information with the DB2 optimizer. In addition, we also added a set of built-in DB2 functions and related query rewrite support to allow for the most user-friendly formulation of SQL queries with full text search.

The remainder of this paper focuses on these features, which were added to DB2 Universal Database Version 7.2 (DB2 henceforth) in support of IBM’s DB2 Text Information Extender (TIE) [2], the successor to TE. In addition to adding support for the exchange of information with the DB2 optimizer, TIE exploits IBM’s next generation text search engine GT9, the successor to TSE. GT9 technology is also used in many other IBM products such as Lotus Notes, Enterprise Information Portal, and Intelligent Miner for Text.

## 2 DB2 Optimization in Support of Full Text Search

The DB2 SQL compiler is a direct descendent of Starburst [3] and generates a query execution plan similarly via parsing, semantics, query rewrite [4], and cost-based optimization phases. The following subsections describe changes made to the compiler to support full text search.

### 2.1 Semantics

DB2 provides 3 scalar functions in support of full text search. Each operates on a column and text search string.

- CONTAINS returns 1 if the text search string matches the first argument, and 0 otherwise.
- SCORE returns a float value between 0 and 1 indicating how well the text search condition was matched.
- NUMBEROFMATCHES returns an integer indicating how often it matched.

These functions serve only a semantic role. There is no supporting DB2 implementation. As will be discussed in the next subsection, a reference to these functions triggers a transformation to an equivalent query involving a table function supplied by TIE as a gateway to GT9.

This table function, which we call *text_search* for this exposition, takes as input meta-data identifying the text index that will support the text search, a text search string, a plan for evaluating additional parts of the query, and any input values required to evaluate the plan. The function returns the primary keys of all documents matching the text search condition as well as the columns *contains*, *score*, and *numberofmatches*, which supply the results for the respective DB2 scalar functions described above.

### 2.2 Query Rewrite

A scalar function interface provides the most user-friendly way to integrate text search, but a naive implementation could not exploit text indexes and would be very inefficient. The query rewrite component handles the burden of resolving TIE meta-data and composing joins with the *text_search* table function. To illustrate the basic idea, consider the following query that determines how relevant 2001 articles are with respect to the subject “G.W. Bush”.

```
SELECT title, SCORE (a.abstract, ’”G.W. Bush”’ )
FROM articles a
WHERE a.year = 2001
```

Upon encountering the *score* function reference, DB2 retrieves TIE meta-data for the *abstract* column, resolves the primary key of the *articles* table, and transforms the query into the following:

```
SELECT a.title, COALESCE(b.score, 0)
FROM articles a LEFT JOIN text_search (‘abstract_meta-data’, ’”G.W. Bush”’, ....) b ON a.isbn = b.key
WHERE a.year = 2001
```

A left join operator is required to assure that 2001 articles not matching the text search condition are preserved. The coalesce function provides these rows with a relevance score of 0.

As illustrated with the following example, the rewrite logic assures that multiple scalar functions are mapped to the same table function output when possible:

```
SELECT title, SCORE (a.abstract, ’”G.W. Bush”’ )
FROM articles a
WHERE a.year = 2001 AND CONTAINS (a.abstract, ’”G.W. Bush”’) = 1
```

The rewrite logic first processes the *score* function reference as in the previous example. It then encounters the reference to the *contains* function, and after comparing its corresponding index meta-data and text search strings with those of the existing table function, determines it can simply be mapped to the output of that function as follows:

```
SELECT a.title, COALESCE(b.score, 0)
FROM articles a LEFT JOIN text_search (‘abstract_meta-data’, ’”G.W. Bush”’, .....) b ON a.isbn = b.key
WHERE a.year = 2001 AND b.contains = 1
```

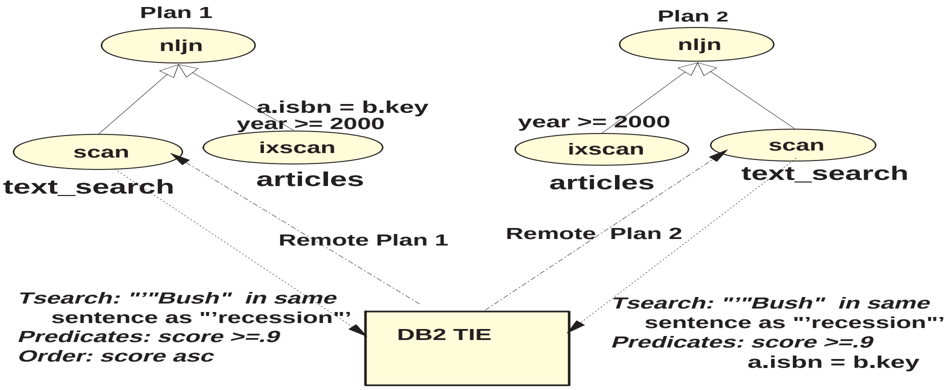
No COALESCE function is needed in this case since the optimizer’s theorem-prover can determine that the conjunct *COALESCE(b.contains, 0)=1* rejects rows when instantiated with NULL. This is an important point, as it allows the outer join operator to be converted to an inner join operator later, by an existing rewrite rule.

## 2.3 Cost-based Optimization

Each time the DB2 optimizer evaluates an alternative plan for accessing the *text_search* table function, it calls the TIE optimizer to exchange costing and planning information. During each exchange, the DB2 optimizer presents the TIE optimizer with a set of *requirements* it would like TIE to satisfy. Examples of requirements includes the text search condition, predicates on output columns, output order, etc.

The TIE optimizer returns a *remote plan* for satisfying one or more of the requirements. The remote plan is treated as a black box by the DB2 optimizer. It is simply passed back to TIE when the *text_search* table function is opened for execution. A set of *properties* is associated with the plan. The properties describe which of the requirements the remote plan will satisfy (it will at least satisfy the text search condition), the cost of satisfying those requirements, and the cardinality of the result. The DB2 optimizer will add logic to the overall plan to compensate for any requirements not satisfied by TIE.

The figure below illustrates the interaction between the DB2 and the TIE optimizers. Two alternative evaluation plans for the query from Section 1 are shown. In plan 1, the *text_search* table function is accessed on the outer of a nested-loops join operation. In plan 2 the roles are reversed. The figure illustrates the sets of requirements passed to the TIE optimizer in each case. In both cases, the DB2 optimizer passes the text search condition and the predicate *score* $>=$ *.9* as requirements. The DB2 optimizer also includes the order requirement with the plan 1 requirements, since rows produced in score order would propagate through the join operation, satisfying the ORDER BY requirement. Remote plan 1 is passed back to TIE (argument 3) at execution time if plan 1 is cheapest overall. The DB2 optimizer forgoes adding the ordering requirement to the plan 2 requirements, but instead adds the join predicate *a.isbn = b.key*. If the join predicate is accepted by TIE, and plan 2 is cheapest overall, the *text_search* table function will be evaluated for each outer row. In this case, DB2 will pass outer row values needed to evaluate the join predicate (argument 4).



## 3 Related and Future Work

Integration of text search into an RDBMS is often done via some type of index extension mechanism [5], [6]. These approaches fail to exploit the complex query evaluation capabilities of advanced text search engines. For example, GT9 can evaluate predicates, produce ordered results, limit those results to only the first $n$ rows, and so on. Modeling the search engine as a remote data source that participates in query optimization allowed us to exploit these capabilities. In the future, we would like to explore the idea of generalizing and exposing the costing and planning interface so that other text search engines can interact with DB2 in this way.

TIE is well-suited for content management applications that typically issue complex queries combining text and relational search conditions. These queries usually require complete answer sets. In contrast, e-commerce applications issue simpler queries with either no relational search conditions, or with only very simple ones. Moreover, these queries require only a few relevant results. IBM offers a specialized solution for e-commerce scenarios, called Net Search Extender (NSE). NSE exploits main memory techniques to achieve extremely high performance and scalability. TIE and NSE will be integrated into a single product in the future.

## 4 Conclusion

This paper describes DB2 optimization support for queries integrating relational and full text search, the key aspect of which is a costing and planning interface that allows our text search engine to fully participate in determination of the optimal query execution plan. The paper also describes how rewriting improves usability and performance. Special thanks to Laura Haas for reviewing an earlier draft of the paper.

## References

[1] L. Haas, D. Kossmann, E. Wimmers, and J. Yang: *Optimizing Queries Across Diverse Data Sources,* VLDB'97

[2] Text Information Extender: *http://www.ibm.com/software/data/db2/extenders/textinformation*

[3] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh: *Extensible Query Processing in Starburst,* SIGMOD'89

[4] H. Pirahesh, J. Hellerstein, W. Hasan: *Extensible Rule-based Query Rewrite Optimization in Starburst,* SIGMOD'92

[5] J. Srinivasan et al.: *Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i,* ICDE 2000

[6] Informix Dynamic Server V9.3, *Virtual Index Interface, Programmers Manual,* Part No. 000-8345, August 2001

# Microsoft SQL Server Full-Text Search

James R. Hamilton
Microsoft Corporation
JamesRH@microsoft.com

Tapas K. Nayak
Microsoft Corporation
TapasNay@microsoft.com

## 1 Introduction

Over the last decade, the focus of the commercial database management community has been primarily on structured data and the industry as a whole has been fairly effective at addressing the needs of these structured storage applications. However, only a small fraction of the data stored and managed each year is fully structured while the vast preponderance of stored data is either wholly unstructured or only semi-structured in the form of documents, web-pages, spreadsheets, email, and other weakly structured formats.

This work investigates the features and capabilities of the full text search access method in Microsoft SQL Server 2000 and the follow-on release of this product and how these search capabilities are integrated into the query language. We will first outline the architecture for the full text search support, then describe the full text query features in more detail, and finally show examples of how this support allows single SQL queries over structured, unstructured, and semi-structured data.

## 2 SQL Server Search Architecture

The SQL Server full text search feature leverages the same underlying full text search access method and infrastructure employed in other Microsoft products, including Exchange, Sharepoint Portal Server, and the Indexing Service that supports full text search over filesystem hosted data. This approach has several advantages, the most significant of which are 1) common full text search semantics across data stored in relational tables, the mail system, web hosted data, and filesystem resident data, and 2) leverage of full text search access method and infrastructure investments across many complementary products.

Indexed text in SQL Server can range from a simple character string data to documents of many types, including Word, Powerpoint, PDF, Excel, HTML, and XML. The document filter support is a public interface, allowing for custom document formats to be integrated into SQL Server full text search. The architecture is composed of five modules hosted in three address spaces (see Figure 1): 1) content reader, 2) filter daemon, 3) word breaker, 4) indexer, and 5) query processor.

Full text indexed data stored in SQL Server tables is scanned by the content reader which assemble data and related metadata packets. These packets flow to the main search engine, which triggers the search engine filter daemon process to consume the data read by the content reader. Filter daemons are modules managed by MS Search but outside of the MS Search address space. Since the search architecture is extensible and filters may be sourced from the shipped product, ISV supplied or ISV produced and there is a risk that a filter bug or a

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

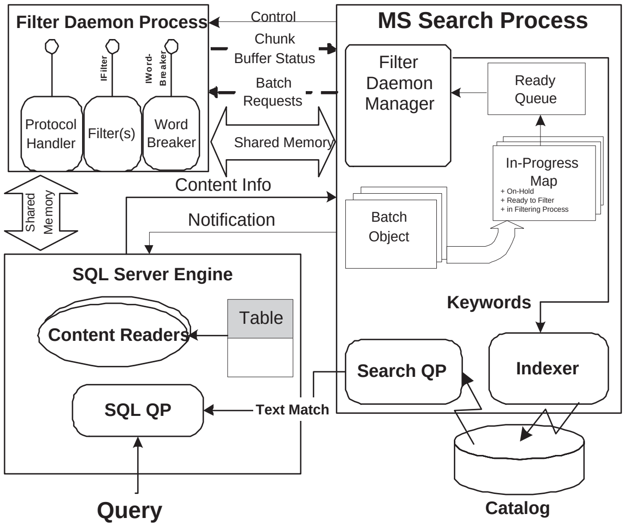


Figure 1: Architecture of SQL Server Full-Text Search.

combination of a poorly formed document and a filter bug could allow the filter to either fail or not terminate. Running the filters and word breakers in an independent process allows the system to be robust in the presence of these potential failure modes. If some instance of daemon fails or runs rogue, MS Search process restarts a new instance.

Filters are invoked by the daemon based on the type of the content. Filters parse the content and emit chunks of processed text. A chunk is a contiguous portion of text along with some relevant information about the text segment like the language-id of the text, attribute information if any, etc. Filters emit chunks separately for any properties in the content. Properties can be items such as title or author and are specific to the content types.

Word breakers break these chunks into keywords. Word breakers are modules that are human language-aware. SQL Server search installs word breakers for various languages, including but not limited to English (USA and UK), Japanese, German, French, Korean, Simplified and Traditional Chinese, Spanish, Thai, Dutch, Italian, and Swedish. The word breakers are also hosted by the filter daemons and they emit keywords, alternate keywords, and location of the keyword in the text. These keywords and related metadata are transferred to the MS Search process via a high speed shared memory protocol that feeds the data into the Indexer. The indexer builds an inverted keyword list with a batch consisting of all the keywords from one or more content items. Once MS Search persists this inverted list to disk, it sends notification back to the SQL Server process confirming success. This protocol ensures that, although documents are not synchronously indexed, documents will not be lost in the event of process or server failures and it allows the indexing process to be restartable based upon metadata maintained by the SQL Server kernel.

As with all text indexing systems we have worked upon, the indexes are stored in a highly compressed form that increases storage efficiency at the risk of driving up the cost of update. To obtain this storage size reduction without substantially increasing update cost, a stack of indexes are maintained. New documents are built into a small index, which is periodically batch merged into a larger index that, in turn, is periodically merged into the base index. This stack of indexes may be greater than three deep but the principle remains the same and it is an engineering approach that allows the use of an aggressively compressed index form without driving up the insertion costs dramatically. When searching for a keyword, all indexes in this stack need to be searched so there is some advantage in keeping the number of indexes to a small number. During insertion and merge operations, distribution and frequency statistics are maintained for internal query processing use and for ranking purposes.

This whole cycle sets up a pipeline involving the SQL Server kernel, the MS Search Engine and the filter daemons, combination of which is the key to reliability and performance of SQL Server full text indexing process.

## 3 SQL Server Full-text Search Query Features

The full text indexes supported by SQL Server are created using the familiar *CREATE INDEX* SQL DDL statement. These indexes are fully supported by SQL Server standard utilities, such as backup and restore, and other administrative operations, such as database attach/detach work unchanged in the presence of full text search indexes. Other enterprise-level features, including shared disk cluster failover, are fully supported in the presence of full text indexes. Indexes are created and maintained online using one of three options: *1) Full Crawl* scans the full table and builds or rebuilds a complete full text index on the indexed columns of the table. This operation proceeds online with utility progress reporting. *2) Incremental Crawl* uses a timestamp column on the indexed table to track changes to the indexed content since the last re-index. *3) Change Tracking* is used to maintain near real time currency between the full text index and the underlying text data. The SQL Server Query Processor directly tracks changes to the indexed data and this data is applied in near real time to the full text index.

The full text search support is exposed in SQL using the following constructs:

1. **Contains Predicate:** *Contains(col_list, '<search condition>').* A contains predicate is true if any of the indicated columns in the list *col_list* contains terms that satisfy the given search condition. A search condition can be a keyword, a keyword prefix, a proximity term, or some combination of these. For example a predicate *Contains(description, ('word* or Excel or "Microsoft Access"'))* will match all entries with description containing words like 'word', 'wordings', 'Excel' or the phrase 'Microsoft Access'.

2. **Freetext Predicate:** Freetext predicates match on text containing terms that are linguistically similar (stemming) to the terms in the search condition. Thus *Freetext(description, 'run in the rain')* will match all the items that contain in its description column text with terms like *run*, *running*, *rain*, *rains*, etc.

3. **ContainsTable and FreetextTable:** ContainsTable and FreetextTable are table-valued functions that locate entries using a search condition as above, and return the matching items along with a rank value for each item computed based on term statistics in the item as well as in the whole corpus.

The search condition for any of the predicates described above can include:

1. Keyword, phrase, prefix: E.g., *Excel, Microsoft Word, word*.*

2. Linguistic generation of relevant keywords: Thesaurus and Inflectional Forms: *Contains(*, 'FORMSOF(INFLECTIONAL, distributed) AND FORMSOF(THESAURUS,databases)')* will find documents containing inflectional forms of *distributed* and all words meaning the same as *databases* (thesaurus support).

3. Weighted Terms: Query terms can be assigned relative weight to impact the rank of matching documents. In the following, the *spread* search term is given twice the weight of the *sauces* search term:

   ```
   SELECT a.CategoryName, a.Description, b.rank
   FROM Categories a, ContainsTable( Categories, description,
      'ISABOUT(spread weight (.8), sauces weight(.4),
      relishes weight(.2))') b
   WHERE a.categoryId = b.[key]
   ```

4. Proximity: One can specify queries using proximity (NEAR) between terms in a matching document, e.g., 'distributed NEAR databases' matches items in which the term *distributed* appears close to *databases*.

5. Composition: Terms A and B can be composed as A AND B, A OR B and A AND NOT B.

## 4 Examples of Full-text Query Scenarios

**Example 1.** We have a table *Documents(DocId, Title, Author, Content)* of documents published in a site. This following query finds all documents authored by Linda Chapman on *child development* and *insomnia* which include the term *child* close to the term *development* in Title. The result is presented in descending order of rank.

```
 SELECT a.Title,b.rank
 FROM Documents a,
    FreetextTable(Documents, Content, '"child development" AND insomnia') b
 WHERE a.DocId=b.[key] and a.Author='Linda Chapman' and
      Contains(a.Title, 'child NEAR development')
 ORDER BY b.rank DESC
```

**Example 2 (Heterogeneous sources).** Data is stored as emails (data source *exchange*), locally-authored documents (data source *monarch*), documents published in SQL Server (schema same as above). The following query gets all documents related to *marketing* and *cosmetics* from all three stores.

```
--Get qualifying email docs
  SELECT DisplayName, hRef FROM openquery(exchange,
      'SELECT "DAV:displayname" as DisplayName, "DAV:href" as hRef
       FROM "manager\Inbox"
       WHERE contains(*, ''marketing AND cosmetics'')')
UNION ALL  --Get qualifying filesystem data
  SELECT filename, vpath  FROM OpenQuery(Monarch,
      'SELECT vpath, Filename
       FROM SCOPE(''deep traversal of "c:\My Documents"'')
       WHERE contains(''marketing AND cosmetics'')')
UNION ALL  --Get qualifying SQL Server data
  SELECT Title, 'SQLServer:Documents:'+cast(DocId as varchar(9)) as href
  FROM Documents
  WHERE contains(*,'marketing AND cosmetics')
```

## 5 Conclusions

In this paper we motivate the integration of a native full text search access method into the Microsoft SQL Server product, describe the architecture of the access method and motivate some of the trade-offs and advantages of the engineering approach taken. We explore the features and functions of the full text search feature and provide example SQL queries showing query integration over structured, semi-structured, and unstructured data.

For further SQL Server 2000 full text search usage and feature details one may look at Inside Microsoft SQL Server [1] or SQL Server 2000 Books online [2]. On the implementation side, we are just completing a major architectural overhaul of the indexing engine and its integration with SQL Server in the next release of the product and this paper is the first description of this work.

## References

[1] Delaney, Kalen. *Inside Microsoft SQL Server 2000,* Microsoft Press, 2001.

[2] *SQL Server 2000 Books Online.*

# Basics of Oracle Text Retrieval

Paul Dixon
Oracle Corporation
paul.dixon@oracle.com

**Abstract**

*This paper covers the basics of text search within the Oracle RDBMS. Emphasis is on demonstrating the simplicity and power of the SQL CONTAINS() text query language.*

## 1 Introduction

For the average user, searching through documents in a file system means using "grep" - this is a linear scan using regular expressions. At a certain size of collection an index becomes necessary. For text this will typically be an "inverted list" index, although other text indexes may be used for specialized applications. An inverted list looks the same wherever the source documents may reside; in a database, on a file system, or on the Web. In Oracle, text indexes are stored natively as database tables.

The primary interface to the relational database is SQL (Structured Query Language), which most people know from handling structured data (e.g., numbers, dates, short text strings, etc.), and the interface to the text-retrieval component is no exception. The primary interface to text is the CONTAINS() function, but CONTAINS() works within the greater framework of the SQL language.

## 2 Defining the Repository

In a file system, a file has disk storage and metadata, whereas in a database a document consists of a database cell (or a set of cells), where a database cell occurs in the individual row and column of a table. A file will generally have an associated system date and other meta information will appear as a file name and security permissions. In the database, metadata is not predefined, but can include system date and arbitrary associated column data.

Database storage is defined in terms of database tables and types:

**Example 1:** CREATE TABLE recipes (name CHAR(30), directions VARCHAR2(400));

This is nearly the simplest possible schema definition (a 30-character-width text field and a 400-character-maximum variable-width field). It could be augmented with other types; structured types such as the date of insertion, the time to create the recipe, cross references, id number, and so on. Text itself can be stored in multiple formats - in a LOB (large object) or as a Binary LOB if it is in a non-text format, such as a word-processing file, or is in a character set other than that of the host database. Other supported datatypes include Web (URL), external file, and procedural (user-defined).

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

## 3 Getting Documents In

The SQL INSERT statement is the most accessible and familiar interface for database data input.

**Example 2:** INSERT INTO recipes VALUES ( ‘omelet’, ‘Finely chop onion, put 1 tbsp. butter in skillet over high flame, mix two eggs, salt and pepper with fork in a bowl, add to skillet, shake and run fork through eggs, when eggs are set, add cheese and onion, fold in half, serve.’);

INSERT is primarily designed for shorter text fragments. Other methods of inserting documents are the Oracle Call Interface (C language), SQL*Loader (a stand-alone specialized data/document loader), and the Oracle Internet File System, which supports file system protocols, as well as FTP and HTTP.

## 4 Finding Information

Creating a full-text index is straightforward:

**Example 3:** CREATE INDEX recipeidx ON recipes(directions) INDEXTYPE IS CONTEXT;

This create index statement works like a “normal” B-tree index. It creates a set of tables that comprise an inverted list index on the text column. The inverted index is effectively the same as the back-of-book index in a paper book; it matches a token or phrase and its locations to speed up search with a look-up rather than a scan.

To query the text column, one uses CONTAINS:

**Example 4:** SELECT name FROM recipes WHERE CONTAINS(directions, ‘cheese’)>0;

Here the SELECT acts upon the full-text index, to return all entries that contain the token ‘cheese’ within the ‘directions’ field. The CONTAINS function can be included in SQL of arbitrary complexity, in conjunction with structured conditions, join conditions, views and sub-selects; it is a row source native to the Oracle RDBMS. When select statements include both full-text and structured components, the optimizer will choose the correct execution plan based on the cost and selectivity of all predicates. It is possible to influence the plan generation for text using optimizer “hints”, the same as for structured conditions.

## 5 Updating the Collection

To add to the indexed document collection, we can insert new documents:

**Example 5:** INSERT INTO recipes VALUES( ‘grilled cheese sandwich’, ‘oil skillet and heat over medium flame, put one slice of cheddar cheese and one of cheese (jack) between bread slices, fry on each side until brown’);

Unlike Oracle B-tree indexes, text indexes need to be explicitly updated. This is to avoid index fragmentation due to the relatively small number of individual token occurrences within a single document.

**Example 6:** ALTER INDEX recipeidx REBUILD ONLINE PARAMETERS(‘SYNC’);

Index synchronization is usually run as a timed job (DBMS_JOB) at an interval based upon the degree of index fragmentation, and the document DML rates. Where batch-inserts are the norm, synchronization will take place immediately following, whereas in an ad-hoc insert environment, SYNC is usually run at regular intervals throughout the day.

# 6 Search Techniques

In its simplest form, CONTAINS can be thought of as a boolean function (ignoring the integer return value). For a token or phrase ("keyword") search this can be perfectly acceptable, particularly on small document collections, or short documents (such as would be found in a product catalog). For larger sets, however, SCORE() can be used to rank the result-set by degree of relevance to the text query.

CONTAINS has its own internal extended boolean text-query syntax, which can be used to augment the text tokens searched for. Below are a few of the most useful features.

## 6.1 Basic

Relevance, for the context index, is defined in terms of term frequency (the more times a term occurs in a document, the higher the relevance), dampened by the total number of occurrences of the term in the collection. If a term is frequent in a document collection, we say it contributes less information content - it is more 'noise'.

The simplest ordered query is on a single token:

**Example 7:** SELECT name, SCORE(1) FROM recipes WHERE CONTAINS(directions, 'cheese', 1)>0 ORDER BY SCORE(1) DESC;

| NAME | SCORE(1) |
|---|---|
| grilled cheese sandwich | 7 |
| omelet | 4 |

Here we can see that the document that mentions 'cheese' more often (twice) gets the higher relevance-ranking. SCORE() is a pseudo-column that carries the relevance information ancillary to the CONTAINS function. The label, '1' in this case, is used to match the SCORE to the appropriate CONTAINS, as there may be several CONTAINS in a single SELECT statement.

## 6.2 Weight (*)

Each query component can be differentiated by 'weighting': for instance the top criterion may be be weighted four times as much as the second, which is weighted twice the weight of the third, in order to give a progressive relaxation of subqueries.

**Example 8:** 'onion * 4 AND cheddar'

Here the 'onion' part of the text query is weighted higher than the 'cheddar' component. All documents that match this query will have both terms, but SCORE will be higher for 'onion' unless it is greatly outnumbered by the other term.

## 6.3 Phrase

Phrase searches find an exact match. 'cheddar cheese' finds only those two words in sequence and next to each other.

## 6.4 NEAR (;)

NEAR is similar to 'phrase' but tokens only need to be close to each other, not next to each other - it is a kind of less-severe proximity search. Words can appear in any order. Similar to phrase, NEAR has a linguistic aspect to it, which is loosely defined; such a definition would be something like "two words occurring in the same sentence" or "two words occurring in the same paragraph."

**Example 9:** ‘jack ; cheese’

This query finds ‘cheese (jack),’ which would be missed by a phrase-only search. It would, however, find ‘jack and jill, mouse cheese,’ which might not be desired.

### 6.5 Accumulate

Accumulate always assigns a higher SCORE to a document matching any $N$ query components than to a document matching only any $N-1$ components. Even though all components may not occur in a document, however, SCORE will be positive if individual terms do occur. This gives high query relevance at the top of a score-sorted result set, while allowing a longer ‘tail’, to reduce the chances of an empty result.

**Example 10:** ‘cheese , swiss , cheddar’

This query will match both example documents, even though only two of the terms are found. The ‘grilled cheese sandwich’ document, matching two query terms, will score higher than the ‘omelet’ document, which matches only one term.

## 7 Further Topics

Other text query operators available to the out-of-the-box Oracle Text installation are boolean AND, OR, NOT, and MINUS, a fuzzy interface to find spelling and OCR errors, and a complete thesaurus infrastructure. Section searching is available to exploit document structure in HTML or XML marked-up documents. A highlighting interface marks up the query terms where they appear in a document, and converts popular word-processor formats to HTML.

Oracle Text can also produce a ‘theme index’[1]. This is an inverted-list index of concepts derived from the document collection and a 425,000-term knowledge base. The system has also been used to build automatic classification and question-answering systems[2].

## References

[1] K. Mahesh, J. Kud, and P. Dixon. *Oracle at TREC8: A Lexical Approach*. Proceedings of the Eighth Text Retrieval Conference, 1999.

[2] S. Alpha, P. Dixon, C. Liao, and C. Yang. *Oracle at TREC10.* Notebook paper, 2001; available at `http://trec.nist.gov/pubs/trec10/papers/orcltrec10.pdf`.

# Structured and Unstructured Search in Enterprises

Prabhakar Raghavan
Verity, Inc.
pragh@verity.com

## 1 Introduction

It is estimated that about a third of the time of a typical enterprise knowledge worker is spent searching for information. Such search and its derivative information retrieval functions are essential components of the infrastructure of *enterprise information portals*, which are the primary means through which enterprise employees access information throughout their business. We begin by listing some essential functions of such software, pointing out where appropriate how such functionality differs from that found in current *web* (as opposed to enterprise) portals that most readers may be familiar with.

1. The need to access information in diverse repositories including file systems, web servers, Lotus Notes, Microsoft Exchange, content management systems such as Documentum, as well as relational databases. In contrast, most web portals deal only with html content.

2. The need to respect fine-grained individual access control rights, typically at the document level; thus two users issuing the same search/navigation request may see differing sets of matching results due to the differences in their privileges. In contrast, most web services have only a single class of access control, or in some cases access levels that form a strict subset hierarchy (as in subscription services with multiple levels of subscription).

3. The need to index and search a large variety of document types (formats), such as PDF, Microsoft Word and Powerpoint files, etc., in many different languages.

4. The need to seamlessly and scalably combine structured (e.g., relational) as well as unstructured information in a document for enhanced navigation and discovery paradigms.

5. The need to combine search results from internal as well as external sources of information.

6. The integration of search with functions like taxonomy building, classification and personalization.

## 2 Verity's K2 Enterprise

We briefly review how these features are addressed in Verity's flagship K2 Enterprise product. We touch upon classification and personalization in this section, deferring a detailed discussion of structured versus unstructured search to Section 3. For each type of repository that is to be accessed (e.g., Documentum, Lotus Notes, databases through ODBC interfaces), Verity provides a *gateway*. The role of the gateway is to allow a spider to access the content in the repository, together with the associated security information (i.e., which users can access which documents). Each repository may contain documents in many different file formats; K2 Enterprise provides

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

*filters* to automatically detect and generate text versions from over 250 document formats. Using gateways and filters, the spider generates a full-text index that consists of two parts: (1) a positional full-text index that records, for each word, every occurrence in each document — such fine-grained positional information is necessary to handle phrase and proximity queries (e.g., *IBM* within 4 words of *Armonk*); (2) a structured index that records, for each document, the meta-data associated with it (much as a database table does).

We now highlight three specific capabilities likely to be of interest to this readership.

1. Arbitrary combinations of text and range queries. Consider the query: *Give me documents that contain the phrase "web server", are of type "Design document", language "German", format "sgml", price* $< 10000$. Clearly this query has a text/unstructured component (to match the phrase "web server") as well as a structured component. We discuss how Verity handles this type query in Section 3, and how this would differ from a solution that might use an RDBMS.

2. Automatically classifying documents by textual content as well as structure derived from meta-data. Adding structure to unstructured information is key to enhancing the value of enterprise content. Classification is a key step in such structure enhancement. This is briefly discussed in Section 2.1.

3. Delivering personalized views of information by re-ranking search results, recommending documents, and locating experts on the query subject. Both explicit profile information and implicit behavioral patterns are exploited in the personalization engine. This is briefly discussed in Section 2.2.

## 2.1 Automatic Classification

While searching provides an efficient way for users to find relevant information in business portals if they know what to search for, there is a different need for browsing and navigating information. Taxonomies are the most popular way for organizing documents into navigable structures. With a taxonomy, users can easily navigate through the category hierarchy to find relevant information. Scoped search within a category typically produces more relevant results than un-scoped search.

There are two main phases in deploying taxonomies in a portal environment: taxonomy construction, and maintenance. The former defines the category hierarchy, and the latter populates documents into the taxonomy once it is built, and modifies the taxonomy structure when needed. Verity refers to a populated taxonomy as a *knowledge tree* which consists of a taxonomy of browsable categories and databases that store the relationships of documents and categories. Fully automatic taxonomy construction and maintenance often leads to unsatisfactory results. Consequently, most taxonomies are built and maintained manually by human experts (examples include the directory structures of Yahoo! and the Open Directory Project). However, manual population of a large volume of documents is expensive.

Verity's Intelligent Classifier [5] provides several taxonomy construction methods, each of which can also be used to categorize documents into a taxonomy: (1) Construct a taxonomy manually through its graphical user interface. (2) Extract a hierarchy of categories of a taxonomy from URL paths or file paths. (3) Fetch categories from indexed meta data. This method can be used when categorical information is explicitly listed in a field in the collection. (4) Generate a taxonomy from document clusters produced by Verity's clustering algorithm, which groups similar documents into clusters. A hierarchy of clusters are generated by recursively breaking large clusters into smaller clusters.

Verity Intelligent Classifier provides the following methods for defining classification rules for each category in the taxonomy. (1) Manually construct classification rules using the Verity Query Language (VQL); (2) Learn classification rules for categories automatically from exemplary documents associated with these categories in the taxonomy; (3) Interactively refine classification rules by providing relevance feedback to the test results of previously built rules. The core technology that supports these automatic features is Verity's regularized Logistic Regression Classifier (LRC). LRC automatically learns a classification rule from a set of documents that are labelled as relevant or irrelevant to a category. Let a document be represented by a feature vector $\mathbf{x} = [t_1, t_2, \cdots, t_d]^T$ , and $r$ be the relevancy measure of the document with respect to the topic, $0.0 \leq r \leq 1.0$. Given a set of relevant documents and a set of irrelevant documents for a category, the LRC learning algorithm learns a regression function $log(r/(1-r)) = w_1 t_1 + w_2 t_2 + \ + w_d t_d + b = f(\mathbf{w}, \mathbf{x})$, such that the separation between the relevant and irrelevant documents is maximized. Structure risk minimization theory [6] guarantees a minimized upper bound on the classification error on future documents.

Once the regression function is determined, a future document can be assigned to the category if the relevancy score $r$ for the document is greater than a pre-specified threshold. This classification rule can be conveniently represented by Verity's powerful query language. Our experiment shows that on Reuter's benchmark data set, LRC achieves the state-of-the-art performance at 88% precision-recall break-even rate [1].

Once a knowledge tree is defined in Verity Intelligent Classifier, it can be exported to Verity Knowledge Organizer [2]. Administrators can configure how to display search results and the category structure to end users. End-users can browse through the documents in a collection by category and drill down to a subject of interest. They can also limit the scope of their searches only to the categories of interest.

### 2.2 Personalization

The Verity Personalization Engine enhances the end-user's information discovery experience to the next level beyond search and taxonomies. It uses a tensor space model that represents different entities in the system such as products, documents, users and queries as tensors in a tensor space. These vectors adapt to latent patterns in user behavior in order to dynamically personalize the results of subsequent searches in different ways, as outlined below. The engine builds and maintains a profile for each user, based on a notion of a *transaction* that updates the profile. Some examples of transactions are:

User <**joe**> bought <**Toaster X**> upon query <**"toaster"**>

User <**john**> viewed <**Document Y**> upon query <**"personalization"**>

The user profiles are used to support the following features: (1) Adaptive Ranking – repeated selection of an item for a given query causes the relevance of the item to be boosted for all users on similar queries. (2) Document Recommendation – products/documents are recommended based on a combination of the current query and the user information, based on the user's past behavior. (3) Document Similarity – documents or products that are similar to a selected item are recommended. (4) Expert Location – based on a document that is being viewed, or a query that has been issued, experts/users in the organization are recommended. (5) Community – a dynamic user community can be presented, based on the current user's profile.

## 3 Text Versus Structured Queries

We begin by highlighting some of the issues in text versus structured search in the enterprise. We mentioned earlier that the index used by a typical enterprise search engine included support for structure as well as unstructured search. We first address the question: how is the structured information typically derived? There are three principal means: (1) from meta-data supplied with the document, say in XML form; (2) using classification techniques such as those described in Section 2.1; (3) by spidering a relational database. In Section 3.1 below we discuss some settings in which it is advantageous to search structured information outside an RDBMS. We pause to highlight a second aspect here: spidering an RDBMS is most useful in the case of, say, catalog information in a B2C or B2B setting. However, most deployments of relational databases entail an *enterprise application* built on top of the database – for supply-chain or customer relationship management, as examples. In these cases, simply spidering the database loses sight of the business logic embedded in the application. Vendors are now attempting to build "application gateways" that spider *through the application*, in the hope of preserving some of the business logic. Such efforts still appear to be in their early stages.

### 3.1 Combined Text and Structured Querying

In the most general settings, each document has some unstructured text as well as a number of structured attributes. Some of these attributes may assume numerical values; for instance, a description of an automobile might have the year in which the car is manufactured and its price. A typical query is a conjunction of an arbitrary text query with an arbitrary range query. In many applications, it is desirable to retrieve and rank documents that simultaneously meet both the unstructured and the structured query components, without recourse to two retrieval systems. Using an RDBMS to solve the problem would result in query responses that would be unacceptably poor. Furthermore, an RDBMS cannot support many features of a full-text engine such as query-time spell correction and proximity searching (mentioned earlier). (Needless to say, this solution does not provide for the numerous other features a typical RDBMS provides, such as logging, recovery, etc.) In addition to retrieving documents that meet the text and parametric queries in a scalable fashion, it is important to be able to rank the results not only by text query scores, but also by sorting along field values (e.g., price). This allows for the efficient navigation of a results list, allowing the user to further refine (or relax) the query at hand.

Our solution consists of augmenting a full-text index with an auxiliary parametric index that allows for fast search, navigation and ranking of query results. We now describe the elements of this index, and how they are organized in order to support all of the operations described above. Each field is represented as a set (known as a BucketSet) and each unique item in that field is represented as a member of that set along with position and frequency information (this datum is known as a Bucket). A set of BucketSet structures make up a completed structure known as the parametric index which maps directly to the corpus from which it was generated. Extra meta-data are stored along with the BucketSets to allow ranges (either text, numeric or date) to be extracted.

It is the mechanism by which the BucketSets and Buckets interact (through their parametric representation) that allows complex set operations (based on nested intersects and unions) to be applied to them. This mechanism results in: (a) real time cardinality statistics for these operations, and (b) an iterative refinement of the operations applied to the database, which can result in the reduction of old Buckets and the inclusion of new Buckets (depending on the semantics applied). In the simple case where A and B are both Buckets of the same set, "$A \bigcap B$" can remove items, whilst "$A \bigcup B$" can add items.

We can also express the text search in the parametric domain (i.e., a text search maps to a BucketSet where each item (hit) is considered as a separate unique Bucket in that set with no context until applied to a parametric index). Both kinds of BucketSets can interact with the same set of operators. Extra speed is obtained by performing query optimizations in real time – for instance, by examining the relative sizes and complexities of the BucketSets when intersecting them and ensuring that the smallest or least complex controls the query optimization. The text search component is built on Verity's VDK kernel [3] with a K2 search engine on top [4]. The approach has been shown to scale to millions of documents with 6-10 fields of structured attributes. Interactive performance is feasible for several concurrent users, using a standard desktop PC as the server.

## References

[1] S.T. Dumais, J. Platt, D. Heckerman and M. Sahami. Inductive learning algorithms and representations for text categorization. *Proc. ACM CIKM*, 1998.

[2] Verity, Inc. *Knowledge Organizer*, v. 2.0, 2000.

[3] Verity, Inc. *Verity Developer Kit (VDK): APR Reference Guide.* v. 3.1, 2000.

[4] Verity, Inc. *K2 Toolkit: API Reference Guide*, v. 2.0, 1999.

[5] Verity, Inc. *Intelligent Classifier*, v. 2.6. 2001.

[6] V. Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, 1995.

# Indexing Methods for Approximate String Matching

Gonzalo Navarro[*] Ricardo Baeza-Yates[*] Erkki Sutinen[†] Jorma Tarhio[‡]

## Abstract

*Indexing for approximate text searching is a novel problem that has received significant attention because of its applications in signal processing, computational biology, and text retrieval, to name a few. We classify most indexing methods in a taxonomy that helps understand their essential features. We show that the existing methods, rather than being completely different as they are often regarded to be, form a range of solutions, whose optimum is usually found somewhere in between the two extremes.*

## 1 Introduction

Approximate string matching is about finding a pattern in a text where the pattern, the text, or both have suffered some kind of undesirable corruption. This has a number of applications, such as retrieving musical passages similar to a sample, finding DNA subsequences after possible mutations, or searching text under the presence of typing or spelling errors.

The problem of *approximate string matching* is formally stated as follows: given a long text $T_{1...n}$ of length $n$ and a comparatively short pattern $P_{1...m}$ of length $m$, both sequences over an alphabet $\Sigma$ of size $\sigma$, find the text positions that match the pattern with at most $k$ "errors".

Among the many existing error models we focus on the popular *Levenshtein* or *edit distance*, where an error is a character insertion, deletion or substitution. That is, the distance $d(x, y)$ between two strings $x$ and $y$ is the minimum number of such errors needed to convert one into the other, and we seek for text substrings that are at distance $k$ or less from the pattern. Most of the techniques can be easily adapted to other error models. We use $\alpha = k/m$ as the error ratio, so $0 < \alpha < 1$.

There are numerous solutions to the *on-line* version of the problem, where the pattern is preprocessed but the text is not [15]. They range from the classical $O(mn)$ worst-case time to the optimal $O((k + \log_\sigma m)n/m)$ average case time. Although very fast on-line algorithms exist, many applications handle so large texts that no on-line algorithm can provide acceptable performance.

An alternative approach when the text is large and searched frequently is to preprocess it: build a data structure on the text (an *index*) beforehand and use it to speed up searches. Many such *indexing methods* have been developed for *exact* string matching [1], but only one decade ago doing the same for *approximate* string matching was an open problem [2].

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

[*] Dept. of Computer Science, University of Chile. Supported in part by Fondecyt grant 1990627.

[†] Dept. of Computer Science, University of Joensuu, Finland.

[‡] Dept. of Computer Science and Engineering, Helsinki University of Technology, Finland.

During the last decade, several proposals to index a text to speed up approximate searches have been presented. No attempt has been made up to now to show them under a common light. This is our purpose. We classify the existing approaches along two dimensions: data structure and search method.

Four different data structures are used in the literature. They all serve roughly the same purposes but present different space/time tradeoffs. We mention them from more to less powerful and space demanding. *Suffix trees* permit searching for any substring of the text. *Suffix arrays* permit the same operations but are slightly slower. *$Q$-grams* permit searching for any text substring not longer than $q$. *$Q$-samples* permit the same but only for some text substrings.

On the other hand, there are three search approaches. *Neighborhood generation* generates and searches for, using an index, all the strings that are at distance $k$ or less from the pattern (their neighborhood). *Partitioning into exact searching* selects pattern substrings that must appear unaltered in any approximate occurrence, uses the index to search for those substrings, and checks the text areas surrounding them. Assuming that the errors occur in the pattern or in the text leads to radically different approaches. *Intermediate partitioning* extracts substrings from the pattern that are searched for allowing fewer errors using neighborhood generation. Again we can consider that errors occur in the pattern or in the text.

Table 1 illustrates this classification and places the existing schemes in context.

| Data Structure | Search Approach | | | | |
|---|---|---|---|---|---|
| | Neighborhood Generation | Partitioning into Exact Searching | | Intermediate Partitioning | |
| | | Errors in Text | Errors in Pattern | Errors in Text | Errors in Pattern |
| Suffix Tree | [10] Jokinen & Ukkonen 91<br>[24] Ukkonen 93<br>[5] Cobbs 95 | | [19] Shi 96 | | |
| Suffix Array | [7] Gonnet 88 | | | | [17] Navarro & Baeza-Yates 99 |
| Q-grams | n/a | [10] Jokinen & Ukkonen 91<br>[9] Holsti & Sutinen 94 | [16] Navarro & Baeza-Yates 97 | | [14] Myers 90 |
| Q-samples | n/a | [21] Sutinen & Tarhio 96 | n/a | [18] Navarro et al. 2000 | n/a |

Table 1: Taxonomy of indexes for approximate text searching. A “n/a” means that the data structure is unsuitable to implement that search approach because not enough information is maintained.

# 2 Basic Concepts

## 2.1 Suffix Trees

Suffix trees [1] are widely used data structures for text processing. Any position $i$ in a text $T$ defines automatically a *suffix* of $T$, namely $T_{i\ldots}$. A *suffix trie* is a trie data structure built over all the suffixes of $T$. Each leaf node points to a suffix. Each internal node represents a unique substring of $T$ that appears more than once. Every substring of $T$ can be found by traversing a path from the root, possibly continuing the search directly in the text if a leaf is reached. In practice a *suffix tree*, obtained by compressing unary trie paths, is preferred because it yields $O(n)$ space and $O(n)$ construction time [12, 25] and offers the same functionality. Figure 1 illustrates a suffix trie.

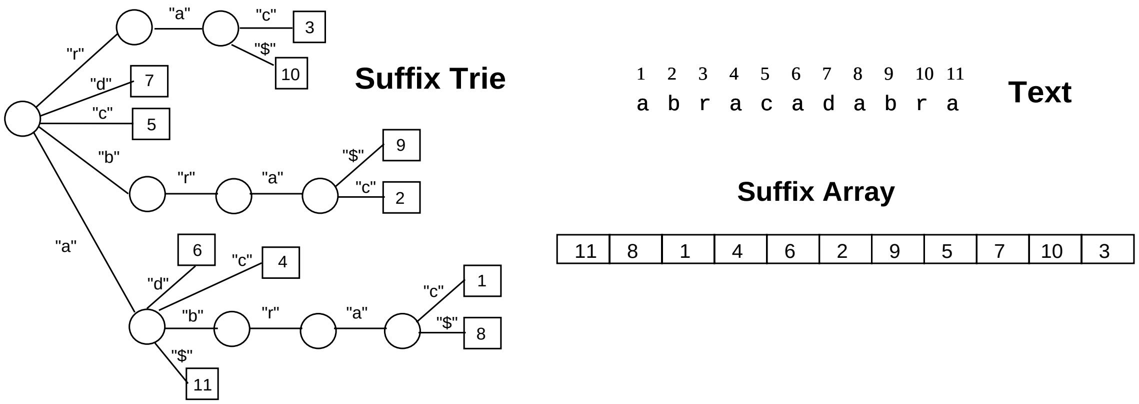


Figure 1: The suffix trie and suffix array for a sample text. The “$” is a special marker to denote the end of the text and is lexicographically smaller than the other characters.

To search for a simple pattern in the suffix trie, we just enter it driven by the letters of the pattern, reporting all the suffix start points in the subtree of the node we arrive at, if any. E.g., consider searching for `"abr"` in the example. So the search time is the optimal $O(m)$. A weak point of the suffix tree is its large space requirement, worsened by the absence of practical schemes to manage it in secondary memory. Among the many attempts to reduce this space, the best practical implementations still require about 9 times the text size [6] and do not handle well secondary memory.

### 2.2 Suffix Arrays

The suffix array [11, 8] is a weak version of the suffix tree, which requires much less space (one pointer per text position, i.e., about 4 times the text size) and poses a small penalty over the search time.

If the leaves of the suffix tree are traversed in left-to-right order, all the text suffixes are retrieved in lexicographical order. A suffix array is simply such an ordered array containing all the pointers to the text suffixes. Figure 1 illustrates this.

The suffix array can be built directly in $O(n \log n)$ worst case time and $O(n \log \log n)$ average time [11]. For secondary memory, a more practical $O(n^2 \log M \ / \ M)$ time algorithm [8] is preferable, where $M$ is the amount of main memory available.

Suffix arrays can simulate by binary searching almost every algorithm on suffix trees, at an $O(\log n)$ time penalty factor. This is because each suffix subtree corresponds to a suffix array interval, so moving to a child node is equivalent to reducing the current suffix array interval by doing two binary searches. For instance, exact searching for a pattern takes $O(m \log n)$ time using this approach.

### 2.3 Q-gram and Q-sample Indexes

Yet a weaker (and less space demanding) scheme is to limit the length of the strings that can be directly found in the index. A $q$-gram index allows retrieval of text strings of length at most $q$.

In a $q$-gram index, every different text $q$-gram (substring of length $q$) is stored. For each $q$-gram, all its positions in the text (called *occurrences*) are stored in increasing text order.

An even less space demanding alternative is a $q$-sample index, where only *some* text $q$-grams (called text $q$-samples) are stored, and therefore not any text $q$-gram can be found. The text $q$-samples, unlike the text $q$-grams, do not overlap, and there may even be some space between each pair of samples. This severely restricted index is attractive for its low space requirements, and it still permits searching for long strings, as we will see later.

A $q$-gram or $q$-sample index can be built in linear time, although for large texts a more practical $O(n \log(n/M))$ time algorithm can be used. Depending on $q$ the index takes from $0.5$ to $3$ times the text size for reasonable retrieval performance.

### 2.4 Computing Edit Distance

The basic algorithm to compute the edit distance between two strings $x$ and $y$ is based on dynamic programming (see [15]). To compute $d(x, y)$ a matrix $C_{0\ldots|x|,0\ldots|y|}$ is filled, where $C_{j,i} = d(x_{1\ldots j}, y_{1\ldots i})$. This is computed as $C_{0,0} = 0$ and

$$C_{j,i} \;=\; \min(C_{j-1,i-1} + \delta(x_j, y_i), C_{j-1,i} + 1, C_{j,i-1} + 1)$$

where $\delta(a, b)$ is zero for $a = b$ and 1 otherwise, and $C_{-1,i} = C_{j,-1} = \infty$. The minimization accounts for the three allowed operations: substitutions, deletions and insertions. At the end, $C_{|x|,|y|} = d(x, y)$. The matrix is filled, e.g., column-wise to guarantee that necessary cells are already computed. The table in Figure 2 (left) illustrates this algorithm to compute $d($`"survey"`, `"surgery"`$)$.

The algorithm is $O(|x||y|)$ time in the worst and average case. The space required is only $O(|x|)$ in a column-wise processing because only the previous column must be stored to compute the new one.

## 3 Neighborhood Generation

### 3.1 The Neighborhood of the Pattern

The number of strings that match a pattern $P$ with at most $k$ errors is finite, as the length of any such string cannot exceed $m + k$. We call this set of strings the “$k$-neighborhood” of $P$, and denote it $U_k(P) = \{x \in \Sigma^*,\ d(x, P) \leq k\}$.

The idea of this approach is, in essence, to generate all the strings in $U_k(P)$ and use an index to search for their text occurrences (without errors). Each such string can be searched for separately, as in [14], or a more sophisticated technique can be used (see next).

The main problem with this approach is that $U_k(P)$ is quite large. Good bounds [23, 14] show an exponential growth in $k$, e.g., $|U_k(P)| = O(m^k \sigma^k)$ [23]. So this approach works well for small $m$ and $k$.

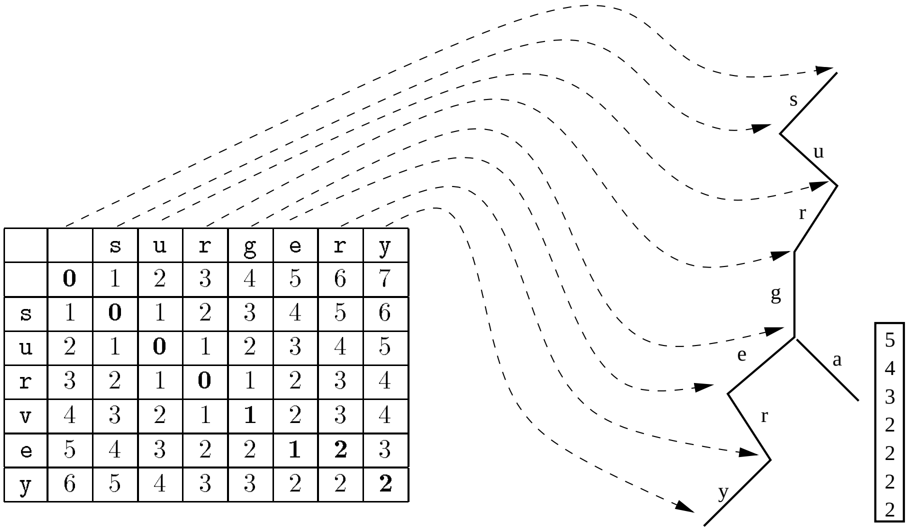
### 3.2 Backtracking

The suffix tree or array can be used to find all the strings in $U_k(P)$ that are present in the text [7, 24]. Since every substring of the text (i.e., every potential occurrence) can be found by traversing the suffix tree from the root, it is sufficient to explore every path starting at the root, descending by every branch up to where it can be seen that that branch cannot start a string in $U_k(P)$.

We explain the algorithm on a suffix trie. We compute the edit distance between our pattern $x = P$ and every text string $y$ that labels a path from the root to a trie node $N$. We start at the root with the initial column $C_{j,root} = j$ (Section 2.4 with $i = 0$) and recursively descend by every branch of the trie. For each edge traversed we compute a new column from the previous assuming that the new character of $y$ is that labeling the edge just traversed.

Two cases may occur at node $N$: $(a)$ We may find that $C_{m,N} \leq k$, which means that $y \in U_k(P)$, and hence we report all the leaves of the current subtree as answers. $(b)$ We may find that $C_{j,N} > k$ for every $j$, which means that $y$ is not a prefix of any string in $U_k(P)$ and hence we can abandon this branch of the trie. If none of these two cases occur, we continue descending by every branch. If we arrive at a leaf node, we continue the algorithm of Section 2.4 over the text suffix pointed to.

Figure 2 illustrates the process over the path that spells out the string `"surgery"`. The matrix can be seen now as a stack (that grows to the right). With $k = 2$ the backtracking ends indeed after reading `"surge"` since that string matches the pattern (case $(a)$). If we had instead $k = 1$ the search would have been pruned (case $(b)$) after considering `"surger"`, and in the alternative path shown, after considering `"surga"`, since in both cases no entry of the matrix is $\leq 1$.



| | | s | u | r | g | e | r | y |
|---|---|---|---|---|---|---|---|---|
| | **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| s | 1 | **0** | 1 | 2 | 3 | 4 | 5 | 6 |
| u | 2 | 1 | **0** | 1 | 2 | 3 | 4 | 5 |
| r | 3 | 2 | 1 | **0** | 1 | 2 | 3 | 4 |
| v | 4 | 3 | 2 | 1 | **1** | 2 | 3 | 4 |
| e | 5 | 4 | 3 | 2 | 2 | **1** | **2** | 3 |
| y | 6 | 5 | 4 | 3 | 3 | 2 | 2 | **2** |

Figure 2: The dynamic programming algorithm run over the suffix trie. We show only one path and one additional link.

Some improvements to this algorithm [10, 25, 5] avoid processing some redundant nodes at the cost of a more complex node processing, but their practicality has not been established. This method has been used also to compare a whole text against another one or against itself [3].

## 4 Partitioning into Exact Search

Each approximate occurrence of a pattern contains some pattern substrings that match without errors. Hence, we can derive sufficient conditions for an approximate match based on exact matching of one or more carefully selected pattern pieces. These pieces are searched for without errors, and the text areas surrounding their occurrences are verified for an approximate occurrence of the complete pattern. This technique is called “filtration” [15].

In indexed searching, some kind of index is used to quickly locate the *exact* occurrences of the selected pattern pieces, and a classical on-line algorithm is used for verification. A general limitation of filtration methods is that there is always a maximum error ratio $\alpha$ up to where they are useful, as for larger error levels the text areas to verify cover almost all the text.

A general lemma is useful to unify the many existing variants.

**Lemma 1:** Let $A$ and $B$ be two strings such that $d(A, B) \leq k$. Let $A = A_1 x_1 A_2 x_2 ... x_{k+s-1} A_{k+s}$, for strings $A_i$ and $x_i$ and for any $s \geq 1$. Then, at least $s$ strings $A_{i_1} \ldots A_{i_s}$ appear in $B$. Moreover, their relative distances inside $B$ cannot differ from those in $A$ by more than $k$.

This is clear if we consider the sequence of at most $k$ edit operations that convert $A$ into $B$. As each edit operation can affect at most one of the $A_i$’s, at least $s$ of them must remain unaltered. The extra requirement on relative distances follows by considering that $k$ edit operations cannot produce misalignments larger than $k$.

Two main branches of algorithms based on the lemma exist, differing essentially in whether the errors are assumed to occur in the pattern or in the text.

### 4.1 Errors in the Pattern

This technique is based on the application of Lemma 1 under the setting $P = A$, $x_i = \varepsilon$. That is, the pattern is split in $k+s$ pieces, and hence $s$ of the pieces must appear inside any occurrence. Therefore, the $k+s$ pieces are searched for in the text and the text areas where $s$ of those pieces appear under the stated distance requirements are verified for a complete match.

Using the data structures of Section 2 the time to search for the pieces in the index is $O(m)$ or $O(m \log n)$, but the checking time dominates. The case $s = 1$, proposed in [16], shows an average time to check the candidates of $O(m^2 kn/\sigma^{m/(k+1)})$. The case $s > 1$ is proposed in [19] without any analysis. It is not clear which is better. If $s$ grows, the pieces get shorter and hence there are more matches to check, but on the other hand, forcing $s$ pieces to match makes the filter stricter [19].

Note that, since we cannot know where the pattern pieces can be found in the text, all the text positions must be searchable. The technique described next, instead, works on a $q$-sample index. A disadvantage of this smaller index is that it tolerates lower error ratios.

### 4.2 Errors in the Text

Assume now that the errors occur in the text, i.e., $A$ is an occurrence of $P$ in $T$. We extract substrings of length $q$ at fixed text intervals of length $h \geq q$.

Those $q$-samples correspond to the $A_i$'s of Lemma 1, and the space between $q$-samples to the $x_i$'s. What the lemma ensures is that, inside any occurrence of $P$ containing $k+s$ text $q$-samples, at least $s$ of them appear in $P$ at about the same positions ($\pm k$). Now, for the lemma to hold, we need to ensure that any occurrence of $P$ in $T$ contains at least $k+s$ text $q$-samples, i.e., $h \leq \lfloor (m-k-q+1)/(k+s) \rfloor$.

At search time, all the $m-q+1$ (overlapping) pattern $q$-grams are extracted and searched for in the index of text $q$-samples. When $s$ pattern $q$-grams match in the text at the proper distances, the text area is verified for a complete match. This idea is presented in [21], and earlier versions in [10, 9, 22].

Let us discuss the best value of $q$. We want it to be small to avoid a very large set of different $q$-samples. We want it to be large to minimize the amount of verification. Some analyses [20] show that $q = \Theta(\log_\sigma n)$ is the optimal value. On the other hand, little has been said about the best $s$ value, except that a larger $s$ may trigger fewer verifications.

## 5 Intermediate Partitioning

We present now an approach that lies between the two previous ones. We filter the search by looking for pattern pieces, but those pieces are large and still may appear with errors in the occurrences. However, they appear with *fewer* errors, and therefore we use neighborhood generation to search for them. A new lemma is useful here.

**Lemma 2:** Let $A$ and $B$ be two strings such that $d(A,B) \leq k$. Let $A = A_1 x_1 A_2 x_2 ... x_{j-1} A_j$, for strings $A_i$ and $x_i$ and for any $j \geq 1$. Let $k_i$ be any set of nonnegative numbers such that $\sum_{i=1}^{j} k_i \geq k - j + 1$. Then, at least one string $A_i$ appears with at most $k_i$ errors in $B$.

The proof is easy: if every $A_i$ needs more than $k_i$ errors to match in $B$, then the total distance cannot be less than $(k-j+1)+j = k+1$. Note that in particular we can choose $k_i = \lfloor k/j \rfloor$ for every $i$.

### 5.1 Errors in the Pattern

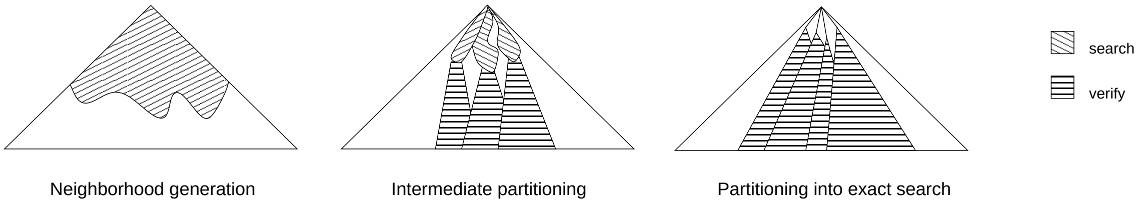
Search approaches based on this method have been proposed in [14, 17]. Split the pattern in $j$ pieces, for some $j$ that we discuss soon. Use neighborhood generation to find the text positions where those pieces appear, allowing

$\lfloor k/j \rfloor$ errors. Then, for each such text position, check with an on-line algorithm the surrounding text. The main question is now which $j$ value to use.

In [14], the pattern is partitioned because they use a $q$-gram index, so they use the minimum $j$ that gives short enough pieces (they are of length $m/j$). In [17] the index can search for pieces of any length, and the partitioning is done in order to optimize the search time.

Consider the evolution of the search time as $j$ moves from 1 (neighborhood generation) to $k+1$ (partitioning into exact search). We search for $j$ pieces of length $m/j$ with $k/j$ errors, so the error level $\alpha$ stays about the same for the subpatterns. As $j$ moves to 1, the cost to search for the neighborhood of the pieces grows exponentially with their length, as shown in Section 3.1. As $j$ moves to $k+1$ this cost decreases, reaching even $O(m)$ when $j = k+1$. So, to find the pieces, a larger $j$ is better.

There is, however, the cost to verify the occurrences too. Consider a pattern that is split in $j$ pieces, for increasing $j$. Start with $j = 2$. Lemma 2 states that every occurrence of the pattern involves an occurrence of at least one of its two halves (with $k/2$ errors), although there may be occurrences of the halves that yield no occurrences of the pattern. Consider now halving the halves ($j = 4$), so we have four pieces now (call them "quarters"). Each occurrence of one of the halves involves an occurrence of at least one quarter (with $k/4$ errors), but there may be many quarter occurrences that yield no occurrences of a pattern half. This shows that, as we partition the pattern in more pieces, more occurrences are triggered. Hence, the verification cost grows from zero at $j = 1$ to its maximum at $j = k+1$. The tradeoff is illustrated in Figure 3.



Neighborhood generation

Intermediate partitioning

Partitioning into exact search

Figure 3: Intermediate partitioning can be seen as a tradeoff between neighborhood generation and partitioning into exact search.

In [17] it is shown that the optimal $j$ is $\Theta(m/\log_\sigma n)$, yielding a time complexity of $O(n^\lambda)$, for $0 \leq \lambda \leq 1$. This is sublinear ($\lambda < 1$) for $\alpha < 1 - e/\sqrt{\sigma}$, a well known limit for any filtration approach [15] (although the $e$ is pessimistic and is replaced by 1 in practice). Interestingly, the same results are obtained in [14] by setting $q = \Theta(\log_\sigma n)$. The experiments in [17] show that this intermediate approach is by far superior to both extremes.

### 5.2 Errors in the Text

This time we consider an occurrence containing a sequence of $j$ $q$-samples, which must be chosen at steps of $h \leq \lfloor (m-k-q+1)/j \rfloor$. By Lemma 2, one of the $q$-samples must appear in the pattern with $\lfloor k/j \rfloor$ errors at most. Moreover, if *every* $q$-sample $i$ appears in the pattern block $Q_i = P_{hi-k..hi+q-1+k}$ with $k_i$ errors, then it must hold that $\sum k_i \leq k$.

This method [21, 18] searches every block $Q_i$ in the index of $q$-samples using backtracking, so as to find the least number of errors to match each text $q$-sample *inside* $Q_i$, using a slight modification to the algorithm of Section 3.2. If a zone of consecutive samples is found whose errors add up to at most $k$, the area is verified with an on-line algorithm.

To permit efficient neighborhood searching, we need to limit the maximum error level allowed. Permitting $q$ errors may be too expensive, as every text $q$-sample will be considered. Rather, we choose $q > e \geq \lfloor k/j \rfloor$ and assume that every text $q$-sample indeed matches with $e+1$ errors. We search the pattern blocks permitting only

$e$ errors. Every $q$-sample found with $k_i \leq e$ errors changes its estimation from $e+1$ to $k_i$, otherwise it stays at the optimistic bound $e+1$.

There is a tradeoff here. If we use a small $e$ value, then the search of the $e$-neighborhoods will be cheaper, but as we have to assume that the text $q$-samples not found have $e+1$ errors, some unnecessary verifications will be carried out. On the other hand, using larger $e$ values gives more exact estimates of the actual number of errors of each text $q$-sample and hence reduces unnecessary verifications in exchange for a higher cost to search the $e$-environments.

Not enough work has been done on obtaining the optimal $e$. In [18] it is mentioned that, as the cost of the search grows exponentially with $e$, the minimal $e = \lfloor k/j \rfloor$ can be a good choice. It is also shown experimentally that the scheme tolerates higher error levels than the corresponding partitioning into exact search.

## 6 Conclusions

We have considered indexing mechanisms for approximate string matching, a novel and difficult problem arising in several areas. We have classified the different approaches using two coordinates: the supporting data structure and the search approach. We have shown that the most promising alternatives are those that look for an optimum balance point between exhaustively searching for neighborhoods of pattern pieces and the strictness of the filtration produced by splitting the pattern into pieces.

A separate issue not covered in this paper is indexing schemes for approximate word matching on natural language text. This is a much more mature problem with well-established solutions.

Radically innovative ideas are welcome in this area. One such idea is an approximation algorithm with worst-case performance guarantees [13]. By setting error thresholds, a certain “fuzziness” is tolerated in the results that are produced, so approximation algorithms should be acceptable for most applications. Another novel approach that is starting to receive attention (e.g., [4]) is to use the edit distance to structure the text as a metric space, so our problem is reduced to a metric-space search. More development is necessary to establish how competitive these ideas could be in practice.

## References

[1] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, 1985.

[2] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, 1992.

[3] R. Baeza-Yates and G. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proc. 6th Symp. on String Processing and Information Retrieval (SPIRE’99)*. IEEE CS Press, 1999. Previous version unpublished, Dept. of Computer Science, Univ. of Chile, 1990.

[4] E. Chávez and G. Navarro. A metric index for approximate string matching. In *Proc. 5th Symp. on Latin American Theoretical Informatics (LATIN)*, 2002. Cancun, Mexico. To appear.

[5] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. 6th Ann. Symp. on Combinatorial Pattern Matching (CPM’95)*, LNCS 807, pages 41–54, 1995.

[6] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. 3rd Workshop on Algorithm Engineering (WAE’99)*, LNCS 1668, pages 30–42, 1999.

[7] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zurich, Switzerland, 1992.

[8] G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.

[9] N. Holsti and E. Sutinen. Approximate string matching using $q$-gram places. In *Proc. 7th Finnish Symp. on Computer Science*, pages 23–32. Univ. of Joensuu, 1994.

[10] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science (MFCS'91)*, pages 240–248, 1991.

[11] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, 22(5):935–948, 1993.

[12] E. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23(2):262–272, 1976.

[13] S. Muthukrishnan and C. Sahinalp. Approximate nearest neighbors and sequence comparisons with block operations. In *Proc. ACM Symp. on the Theory of Computing*, pages 416–424, 2000.

[14] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994. Earlier version in Tech. report TR-90-25, Dept. of CS, Univ. of Arizona, 1990.

[15] G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surv.*, 33(1):31–88, 2001.

[16] G. Navarro and R. Baeza-Yates. A practical $q$-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`. Earlier version in *Proc. CLEI'97*.

[17] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms*, 1(1):205–239, 2000. Hermes Science Publishing. Earlier version in *CPM'99*.

[18] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate $q$-grams. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (CPM'2000)*, LNCS 1848, pages 350–363, 2000.

[19] F. Shi. Fast approximate string matching with $q$-blocks sequences. In *Proc. 3rd South American Workshop on String Processing (WSP'96)*, pages 257–271. Carleton University Press, 1996.

[20] E. Sutinen and J. Tarhio. On using $q$-gram locations in approximate string matching. In *Proc. 3rd European Symp. on Algorithms (ESA'95)*, LNCS 979, pages 327–340, 1995.

[21] E. Sutinen and J. Tarhio. Filtration with $q$-samples in approximate string matching. In *Proc. 7th Ann. Symp. on Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, pages 50–61, 1996.

[22] T. Takaoka. Approximate pattern matching with samples. In *Proc. 5th Int'l. Symp. on Algorithms and Computation (ISAAC'94)*, LNCS 834, pages 234–242, 1994.

[23] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[24] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Ann. Symp. on Combinatorial Pattern Matching (CPM'93)*, LNCS 684, pages 228–242, 1993.

[25] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, 1995.

# Using $q$-grams in a DBMS for Approximate String Processing

Luis Gravano
Columbia University
gravano@cs.columbia.edu

Panagiotis G. Ipeirotis
Columbia University
pirot@cs.columbia.edu

H. V. Jagadish
University of Michigan
jag@eecs.umich.edu

Nick Koudas
AT&T Labs–Research
koudas@research.att.com

S. Muthukrishnan
AT&T Labs–Research
muthu@research.att.com

Lauri Pietarinen
ATBusiness Communications
lauri.pietarinen@atbusiness.com

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

## Abstract

*String data is ubiquitous, and its management has taken on particular importance in the past few years. Approximate queries are very important on string data. This is due, for example, to the prevalence of typographical errors in data, and multiple conventions for recording attributes such as name and address. Commercial databases do not support approximate string queries directly, and it is a challenge to implement this functionality efficiently with user-defined functions (UDFs). In this paper, we develop a technique for building approximate string processing capabilities* on top *of commercial databases by exploiting facilities already available in them. At the core, our technique relies on generating short substrings of length $q$, called $q$-grams, and processing them using standard methods available in the DBMS. The proposed technique enables various approximate string processing methods in a DBMS, for example approximate (sub)string selections and joins, and can even be used with a variety of possible edit distance functions. The approximate string match predicate, with a suitable edit distance threshold, can be mapped into a vanilla relational expression and optimized by conventional relational optimizers.*

## 1 Introduction

String data is ubiquitous. To name only a few commonplace applications, consider product catalogs (for books, music, software, etc.), electronic white and yellow page directories, specialized information sources such as patent databases, and customer relationship management data.

As a consequence, management of string data in databases has taken on particular importance in the past few years. However, the quality of the string information residing in various databases can be degraded due

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

to a variety of reasons, including human typing errors and flexibility in specifying string attributes. Hence, the results of operations based on exact matching of string attributes are often of lower quality than expected.

For example, consider a corporation maintaining various customer databases. Requests for correlating data sources are very common in this context. A specific customer might be present in more than one database because the customer subscribes to multiple services that the corporation offers, and each service may have developed its database independently. In one database, a customer's name may be recorded as `John A. Smith`, while in another database the name may be recorded as `Smith, John`. In a different database, due to a typing error, this name may be recorded as `Jonh Smith`. A request to correlate these databases and create a unified view of customers will fail to produce the desired output if exact string matching is used in the join.

Unfortunately, commercial databases do not directly support approximate string processing functionality. Specialized tools, such as those available from Trillium Software[1], are useful for matching specific types of values such as addresses, but these tools are not integrated with databases. To use such tools for information stored in databases, one would either have to process data outside the database, or be able to use them as user-defined functions (UDFs) in an object-relational database. The former approach is undesirable in general. The latter approach is quite inefficient, especially for joins, because relational engines evaluate joins involving UDFs whose arguments include attributes belonging to multiple tables by essentially computing the cross-products of the tables and applying the UDFs in a post-processing fashion.

Although there is a fair amount of work on the problem of approximate string matching (see, for example, [3]), these results are not used in the context of a relational DBMS. In this paper, we present a technique for incorporating approximate string processing capabilities to a database. At the core, our technique relies on using short substrings of length $q$ of the database strings (also known as *$q$-grams*). We show how a relational schema can be augmented to directly represent $q$-grams of database strings in auxiliary tables within the database in a way that will enable use of traditional relational techniques and access methods for performing approximate string matching operations. Instead of trying to invent completely new join algorithms from scratch (which would be unlikely to be incorporated into existing commercial DBMSs), we opted for a design that would require minimal changes to existing database systems. We show how the approximate string match predicate, with a suitable edit distance threshold, can be mapped into a vanilla SQL expression and optimized by conventional optimizers. The immediate practical benefit of our technique is that approximate string processing can be widely and effectively deployed in commercial relational databases without extensive changes to the underlying database system. Furthermore, by not requiring any changes to the DBMS internals, we can re-use existing facilities, like the query optimizer, join ordering algorithms and selectivity estimation.

The rest of the paper, which reports and expands on work originally presented in [2], is organized as follows. In Section 2, we present notation and definitions. In Section 3, we develop a principled mechanism for augmenting a database with $q$-gram tables. We describe the conceptual techniques for approximate string processing using $q$-grams in Section 4. Finally, in Section 5, we show how these conceptual techniques can be realized using SQL queries.

# 2 Preliminaries

## 2.1 Notation

We use $R$, possibly with subscripts, to denote tables, $A$, possibly with subscripts, to denote attributes, and $t$, possibly with subscripts, to denote records in tables. We use the notation $R.A_i$ to refer to attribute $A_i$ of table $R$, and $R.A_i(t_j)$ to refer to the value in attribute $R.A_i$ of record $t_j$. Let $\Sigma$ be a finite alphabet of size $|\Sigma|$. We use lower-case Greek symbols, such as $\sigma$, possibly with subscripts, to denote strings in $\Sigma^*$. Let $\sigma \in \Sigma^*$ be a string of length $n$. We use $\sigma[i \ldots j]$, $1 \leq i \leq j \leq n$, to denote a substring of $\sigma$ of length $j - i + 1$ starting at position $i$.

[1] `www.trillium.com`

To match strings *approximately* in a database, we need to specify the approximation metric. Several proposals exist for strings to capture the notion of "approximate equality." Among them, the notion of *edit distance* between two strings is very popular.

**Definition 1:** The *edit distance* between two strings is the minimum number of edit operations (i.e., *insertions*, *deletions*, and *substitutions* of single characters) needed to transform one string into the other. □

Although we will mainly focus on the *edit distance* metric in this paper, we note that our proposed techniques can be used for a variety of other distance metrics as well.

### 2.2 $Q$-grams: A Foundation for Approximate String Processing

Below, we briefly review the notion of positional $q$-grams from the literature, and we give the intuition behind their use for approximate string matching [7, 6, 4]. Given a string $\sigma$, its *positional $q$-grams* are obtained by "sliding" a window of length $q$ over the characters of $\sigma$. Since $q$-grams at the beginning and the end of the string can have fewer than $q$ characters from $\sigma$, we introduce new characters "#" and "%" *not* in $\Sigma$, and conceptually extend the string $\sigma$ by prefixing it with $q-1$ occurrences of "#" and suffixing it with $q-1$ occurrences of "%". Thus, each $q$-gram contains exactly $q$ characters, though some of these may not be from the alphabet $\Sigma$.

**Definition 2:** A *positional $q$-gram* of a string $\sigma$ is a pair $(i, \sigma[i \ldots i+q-1])$, where $\sigma[i \ldots i+q-1]$ is the $q$-gram of $\sigma$ that starts at position $i$, counting on the extended string. The set $G_\sigma$ of all positional $q$-grams of a string $\sigma$ is the set of all the $|\sigma|+q-1$ pairs constructed from all $q$-grams of $\sigma$. □

The intuition behind the use of $q$-grams as a foundation for approximate string processing is that when two strings $\sigma_1$ and $\sigma_2$ are within a small edit distance of each other, they share a large number of $q$-grams in common [6, 4]. Consider the following example. The positional $q$-grams of length $q$=3 for string `john_smith` are {`(1,##j)`, `(2,#jo)`, `(3,joh)`, `(4,ohn)`, `(5,hn_)`, `(6,n_s)`, `(7,_sm)`, `(8,smi)`, `(9,mit)`, `(10,ith)`, `(11,th%)`, `(12,h%%)`}. Similarly, the positional $q$-grams of length $q$=3 for `john_a_smith`, which is at an edit distance of two from `john_smith`, are {`(1,##j)`, `(2,#jo)`, `(3,joh)`, `(4,ohn)`, `(5,hn_)`, `(6,n_a)`, `(7,_a_)`, `(8,a_s)`, `(9,_sm)`, `(10,smi)`, `(11,mit)`, `(12,ith)`, `(13,th%)`, `(14,h%%)`}. If we ignore the position information, the two $q$-gram sets have 11 $q$-grams in common. Interestingly, only the first five positional $q$-grams of the first string are also positional $q$-grams of the second string. However, an additional six positional $q$-grams in the two strings differ in their position by just two positions each. This illustrates that, in general, the use of positional $q$-grams for approximate string processing will involve comparing positions of "matching" $q$-grams within a certain "band."

## 3 Augmenting a Database with Positional $q$-Grams

To enable approximate string processing in a database system based on the use of $q$-grams, we need a principled mechanism for augmenting the database with positional $q$-grams corresponding to the original database strings.

Let $R$ be a table with schema $(A_0, A_1, \ldots, A_m)$, such that $A_0$ is the key, and some attributes $A_i$, $i > 0$, are string-valued. For each string attribute $A_i$ that we wish to consider for approximate string processing, we create an auxiliary table $RA_iQ(A_0, Pos, Qgram)$ with three attributes. For a string $\sigma$ in attribute $A_i$ of a record of $R$, its $|\sigma|+q-1$ positional $q$-grams are represented as $|\sigma|+q-1$ separate records in the table $RA_iQ$, where $RA_iQ.Pos$ identifies the position of the $q$-gram $RA_iQ.Qgram$. These $|\sigma|+q-1$ records all share the same value for the attribute $RA_iQ.A_0$, which serves as the foreign key attribute to table $R$.

Interestingly, these tables can be created in current database systems, using simple SQL statements. To do so, we use a table $N$ that contains a single attribute $I$ with the numbers from 1 to $M$ (where $M$ is the maximum

```
INSERT  INTO RAiQ
        SELECT R.A0, N.I,
           SUBSTR(SUBSTR('#...#',1,q-1) || UPPER(R.Ai) || SUBSTR('%...%',1,q-1), N.I, q)
        FROM R, N
        WHERE N.I ≤ LENGTH(R.Ai)+q-1;
```

Figure 1: Creating the auxiliary $q$-gram table $RA_iQ$

length of a string) [1]. Then, we join this table with the column $R.A_i$, and we take all the $q$-grams of each string in $R.A_i$ that start at position $x$, where $x$ is the value stored in field $I$ of a tuple of $N$. The result of this join is then used to create the auxiliary table $RA_iQ$. The exact SQL query is presented in Figure 1.

The space overhead for the auxiliary $q$-gram table for a string attribute $A_i$ of a relation $R$ with $n$ records is:

$$S(RA_iQ) = n(q-1)(q+C) + (q+C)\sum_{j=1}^{n} |R.A_i(t_j)|$$

where $C$ is the size of the additional attributes in the auxiliary $q$-gram table (i.e., $A_0$ and $Pos$). Since $n(q-1) \leq \sum_{j=1}^{n} |R.A_i(t_j)|$, for any reasonable value of $q$, it follows that $S(RA_iQ) \leq 2(q+C)\sum_{j=1}^{n} |R.A_i(t_j)|$. Thus, the size of the auxiliary table is bounded by some linear function of $q$ times the size of the corresponding column in the original table.

Depending on the frequency of the approximate string operations, the database administrator can choose whether or not to have the tables permanently materialized. If the space overhead is not an issue, then the cost of keeping the auxiliary tables updated is relatively small. After creating an augmented database with the auxiliary tables for each of the string attributes of interest, we can efficiently perform approximate string processing using simple SQL queries. We describe the methods next.

## 4 Filtering Results Using $q$-gram Properties

In this section, we present our basic techniques for approximate string processing based on the *edit distance metric*. Later we will describe appropriate modifications to these filters to accommodate alternative distance metrics. The key objective here is to efficiently identify candidate answers to our problems by taking advantage of the $q$-grams in the auxiliary database tables and using features already available in database systems such as traditional access and join methods. For reasons of correctness and efficiency, we require *no false dismissals* and *few false positives* respectively.

**Count Filtering:** The basic idea of COUNT FILTERING is to take advantage of the information conveyed by the sets $G_{\sigma_1}$ and $G_{\sigma_2}$ of $q$-grams of the strings $\sigma_1$ and $\sigma_2$, *ignoring positional information*, in determining whether $\sigma_1$ and $\sigma_2$ are within edit distance $k$.

The intuition here is that strings that are within a small edit distance of each other share a large number of $q$-grams in common. This intuition has appeared in the literature earlier [5], and can be formalized as follows.

**Proposition 3:** Consider strings $\sigma_1$ and $\sigma_2$, of lengths $|\sigma_1|$ and $|\sigma_2|$, respectively. If $\sigma_1$ and $\sigma_2$ are within an edit distance of $k$, then the cardinality of $G_{\sigma_1} \cap G_{\sigma_2}$, ignoring positional information, must be at least $(\max(|\sigma_1|, |\sigma_2|) + q - 1) - k * q$. □

Intuitively, this holds because one edit distance operation can modify *at most* $q$ $q$-grams, so $k$ edit distance operations can modify at most $kq$ $q$-grams.

**Position Filtering:** While COUNT FILTERING is effective in improving the efficiency of approximate string processing, it does not take advantage of $q$-gram position information. In general, the interaction between $q$-gram match positions and the edit distance threshold is quite complex. Any given $q$-gram in one string may not occur at all in the other string, and positions of successive $q$-grams may be off due to insertions and deletions. Furthermore, as always, we must keep in mind the possibility of a $q$-gram in one string occurring at multiple positions in the other string.

Intuitively, a positional $q$-gram $(i, \tau_1)$ in one string $\sigma_1$ is said to *correspond* to a positional $q$-gram $(j, \tau_2)$ in another string $\sigma_2$ if $\tau_1 = \tau_2$ and $(i, \tau_1)$, after the sequence of edit operations that convert $\sigma_1$ to $\sigma_2$ and affect *only the position* of the $q$-gram $\tau_1$, "becomes" $q$-gram $(j, \tau_2)$ in the edited string. Notwithstanding the complexity of matching positional $q$-grams in the presence of edit errors in strings, a useful filter can be devised based on the following observation [4].

**Proposition 4:** If strings $\sigma_1$ and $\sigma_2$ are within an edit distance of $k$, then a positional $q$-gram in one *cannot correspond* to a positional $q$-gram in the other that differs from it by more than $k$ positions. □

**Length Filtering:** We finally observe that string length provides useful information to quickly prune strings that are not within the desired edit distance.

**Proposition 5:** If strings $\sigma_1$ and $\sigma_2$ are within edit distance $k$, their lengths cannot differ by more than $k$. □

# 5 Approximate String Processing in a Database

Below we describe how we can use the previously described properties of $q$-grams to perform approximate string processing tasks inside a database system. Additional details, including an experimental evaluation, are presented in [2].

## 5.1 Approximate String Selections

This problem can be formalized as follows: Given a table $R$ with a string attribute $R.A_i$ and a string query $\sigma$, retrieve all records $t \in R$ such that edit_distance$(\sigma, R.A_i(t)) \leq k$.

To perform this operation it is first necessary to create the $q$-gram set for the query string $\sigma$. This can be done easily in SQL, in a manner similar to the SQL statement of Figure 1. These $q$-grams are stored in a small auxiliary table $TQ$. After this step, it is possible to find all the strings in $R.A_i$ that are possible candidate answers. This can be achieved on the augmented database using the SQL statement of Figure 2 that implements the filters described in Section 4. Consequently, if a relational engine receives a request for an approximate string operation, it can directly map it to a conventional SQL expression and optimize it as usual. (Of course, $k$ and $q$ are constants that need to be instantiated before the query is evaluated.) However, even after the filtering steps, the candidate set may still have false positives. Hence, a UDF invocation `edit_distance(`$R.A_i, \sigma, k$`)` still needs to be performed, but hopefully on just a small fraction of the strings.

## 5.2 Approximate String Joins

In a similar manner, we can efficiently implement approximate string joins: given two tables $R_1$ and $R_2$ with string attributes $R_1.A_i$ and $R_2.A_j$ respectively, report all pairs of strings that are within edit distance $k$.

In this case, we directly join the auxiliary $q$-gram tables, and we report pairs of strings with enough corresponding $q$-grams in common. Essentially, the SQL query expression in Figure 3 joins the auxiliary tables corresponding to the string-valued attributes $R_1.A_i$ and $R_2.A_j$ on their $Qgram$ and $Pos$ attributes, along with the foreign-key/primary-key joins with the original database tables $R_1$ and $R_2$ to retrieve the string pairs that need to be returned to the user.

```
SELECT   $R.A_0$, $R.A_i$
FROM     $R$, $TQ$, $RA_iQ$
WHERE    $R.A_0 = RA_iQ.A_0$ AND $RA_iQ.Qgram = TQ.Qgram$ AND
         $RA_iQ.Pos \le TQ.Pos + k$ AND $RA_iQ.Pos \ge TQ.Pos - k$ AND
         LENGTH($R.A_i$) $\le$ LENGTH($\sigma$) $+ k$ AND LENGTH($R.A_i$) $\ge$ LENGTH($\sigma$) $- k$
GROUP BY $R.A_0, R.A_i$
HAVING   COUNT(*)$\ge$ LENGTH($R.A_i$) $- 1 - (k-1)*q$ AND COUNT(*)$\ge$ LENGTH($\sigma$) $- 1 - (k-1)*q$
```

Figure 2: Performing approximate string selections in an augmented DBMS using SQL

```
SELECT   $R_1.A_0$, $R_2.A_0$, $R_1.A_i$, $R_2.A_j$
FROM     $R_1$, $R_1A_iQ$, $R_2$, $R_2A_jQ$
WHERE    $R_1.A_0 = R_1A_iQ.A_0$ AND $R_2.A_0 = R_2A_jQ.A_0$ AND
         $R_1A_iQ.Qgram = R_2A_jQ.Qgram$ AND
         $R_1A_iQ.Pos \le R_2A_jQ.Pos + k$ AND $R_1A_iQ.Pos \ge R_2A_jQ.Pos - k$ AND
         LENGTH($R_1.A_i$) $\le$ LENGTH($R_2.A_j$) $+ k$ AND LENGTH($R_1.A_i$) $\ge$ LENGTH($R_2.A_j$) $- k$
GROUP BY $R_1.A_0, R_2.A_0, R_1.A_i, R_2.A_j$
HAVING   COUNT(*)$\ge$ LENGTH($R_1.A_i$) $- 1 - (k-1)*q$ AND COUNT(*)$\ge$ LENGTH($R_2.A_j$) $- 1 - (k-1)*q$
```

Figure 3: Performing approximate string joins in an augmented DBMS using SQL

### 5.3 Approximate Substring Processing

A different type of approximate string match of interest is based on one string being a substring of another, possibly allowing for some errors. We can formalize the approximate substring selection problem as follows. Given a table $R$ with a string attribute $R.A_i$ and a query string $\sigma$, retrieve all records $t$ from $R$, such that for some substring $\sigma_R$ of $R.A_i(t)$, edit_distance($\sigma_R, \sigma$) $\le k$. For this edit distance metric, we have to revise the filters described in Section 4. Specifically, LENGTH FILTERING and POSITION FILTERING are not applicable, since the $q$-gram at position $i$ in $\sigma$ may match at any arbitrary position in $R.A_i(t)$ and not just in $i \pm k$. Also $R.A_i(t)$ might be of arbitrary length and still have a substring match with $\sigma$. Finally, COUNT FILTERING has a different threshold, reflecting the fact that the $q$-grams at the beginning and at the end of $\sigma$ (with the "extended" characters '#' and '%') might not match the respective $q$-grams of $R.A_i(t)$.

**Proposition 6:** Consider strings $\sigma_1$ and $\sigma_2$. If $\sigma_2$ has a substring $\sigma_S$ such that $\sigma_1$ and $\sigma_S$ are within an edit distance of $k$, then the cardinality of $G_{\sigma_1} \cap G_{\sigma_S}$, ignoring positional information, must be at least $|\sigma_1| - (k+1)q + 1$. □

Using this result, it is possible to write the respective SQL queries to perform selections and joins based on approximate substring matches. The SQL expressions are very similar to the ones described in Figures 2 and 3, but with a different threshold for COUNT FILTERING and without the conditions that perform the POSITION and LENGTH FILTERING.

### 5.4 Allowing for Block Moves

Traditional string edit distance computations are for single character insertions, deletions and substitutions. If a whole block of characters is modified or moved, the cost charged is proportional to the length of the block. In many applications, we would like to keep a fixed charge for block move operations, independent of block length.

It turns out that the $q$-gram method is suited to this enhanced metric, and in this section we consider the issues involved in so doing. For this purpose, we begin by extending the definition of edit distance.

**Definition 7:** The *extended edit distance* between two strings is the minimum cost of edit operations needed to transform one string into the other. The operations allowed are single character insertion, deletion and substitution, at unit cost; and the movement of a block of contiguous characters, at a cost of $\beta$ units. □

**Theorem 8:** Let $G_{\sigma_1}, G_{\sigma_2}$ be the set of $q$-grams for strings $\sigma_1$ and $\sigma_2$ in the database. If the extended edit distance between $\sigma_1$ and $\sigma_2$ is less than $k$, then the cardinality of $G_{\sigma_1} \cap G_{\sigma_2}$, ignoring positional information, is at least $max(|\sigma_1|, |\sigma_2|) - 1 - 3(k-1)q/\beta'$, where $\beta' = min(3, \beta)$. □

Intuitively, the bound arises from the fact that the block move operation can transform a string of the form $\alpha\nu\delta\mu$ to $\alpha\delta\nu\mu$, which can result in up to $3q - 3$ mismatching $q$-grams.

Based on the above observations, it is easy to see that one can apply COUNT FILTERING (with a suitably modified threshold) and LENGTH FILTERING for approximate string processing with block moves. However, incorporating POSITION FILTERING is not possible as described earlier because block moves may end up moving $q$-grams arbitrarily.

Again, it is possible to write the appropriate SQL queries to perform selections and joins based on the extended edit distance. The statements will apply only the correct filters and will return a set of candidate answers than can be later verified for correctness using a suitable UDF.

## 6 Conclusions

The ubiquity of string data in a variety of databases, and the diverse population of users of these databases, has brought the problem of string-based querying and searching to the forefront of the database community. Given the preponderance of errors in databases, and the possibility of mistakes by the querying agent, returning query results based on approximate string matching is crucial. In this paper, we have demonstrated that approximate string processing can be widely and effectively deployed in commercial relational databases without extensive changes to the underlying database system.

## Acknowledgments

L. Gravano and P. Ipeirotis were funded in part by the National Science Foundation (NSF) under Grants No. IIS-97-33880 and IIS-98-17434. The work of H. V. Jagadish was funded in part by NSF under Grant No. IIS-00085945.

## References

[1] Hugh Darwen. A constant friend. In *Relational Database Writings 1985-1989 by C. J. Date*, pages 493–500. Prentice Hall, 1990.

[2] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB 2001)*, pages 491–500, 2001.

[3] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[4] Erkki Sutinen and Jorma Tarhio. On using $q$-gram locations in approximate string matching. In *Proceedings of Third Annual European Symposium on Algorithms (ESA'95)*, pages 327–340, 1995.

[5] Erkki Sutinen and Jorma Tarhio. Filtration with $q$-samples in approximate string matching. In *Combinatorial Pattern Matching, 7th Annual Symposium (CPM'96)*, pages 50–63, 1996.

[6] Esko Ukkonen. Approximate string matching with $q$-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[7] J. R. Ullmann. A binary $n$-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 20(2):141–147, 1977.

# Modern Information Retrieval: A Brief Overview

Amit Singhal
Google, Inc.
singhal@google.com

### Abstract

*For thousands of years people have realized the importance of archiving and finding information. With the advent of computers, it became possible to store large amounts of information; and finding useful information from such collections became a necessity. The field of Information Retrieval (IR) was born in the 1950s out of this necessity. Over the last forty years, the field has matured considerably. Several IR systems are used on an everyday basis by a wide variety of users. This article is a brief overview of the key advances in the field of Information Retrieval, and a description of where the state-of-the-art is at in the field.*

## 1 Brief History

The practice of archiving written information can be traced back to around 3000 BC, when the Sumerians designated special areas to store clay tablets with cuneiform inscriptions. Even then the Sumerians realized that proper organization and access to the archives was critical for efficient use of information. They developed special classifications to identify every tablet and its content. (See `http://www.libraries.gr` for a wonderful historical perspective on modern libraries.)

The need to store and retrieve written information became increasingly important over centuries, especially with inventions like paper and the printing press. Soon after computers were invented, people realized that they can be used for storing and mechanically retrieving large amounts of information. In 1945 Vannevar Bush published a ground breaking article titled “As We May Think” that gave birth to the idea of automatic access to large amounts of stored knowledge. [5] In the 1950s, this idea materialized into more concrete descriptions of how archives of text could be searched automatically. Several works emerged in the mid 1950s that elaborated upon the basic idea of searching text with a computer. One of the most influential methods was described by H.P. Luhn in 1957, in which (put simply) he proposed using words as indexing units for documents and measuring word overlap as a criterion for retrieval. [17]

Several key developments in the field happened in the 1960s. Most notable were the development of the SMART system by Gerard Salton and his students, first at Harvard University and later at Cornell University; [25] and the Cranfield evaluations done by Cyril Cleverdon and his group at the College of Aeronautics in Cranfield. [6] The Cranfield tests developed an evaluation methodology for retrieval systems that is still in use by IR systems today. The SMART system, on the other hand, allowed researchers to experiment with ideas to

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

improve search quality. A system for experimentation coupled with good evaluation methodology allowed rapid progress in the field, and paved way for many critical developments.

The 1970s and 1980s saw many developments built on the advances of the 1960s. Various models of document retrieval were developed and advances were made along all dimensions of the retrieval process. These new models/techniques were experimentally proven to be effective on small text collections (several thousand articles) available to researchers at the time. However, due to lack of availability of large text collections, the question whether these models and techniques would scale to larger corpora remained unanswered. This changed in 1992 with the inception of Text Retrieval Conference, or TREC[1]. [11] TREC is a series of evaluation conferences sponsored by various US Government agencies under the auspices of NIST, which aims at encouraging research in IR from large text collections.

With large text collections available under TREC, many old techniques were modified, and many new techniques were developed (and are still being developed) to do effective retrieval over large collections. TREC has also branched IR into related but important fields like retrieval of spoken information, non-English language retrieval, information filtering, user interactions with a retrieval system, and so on. The algorithms developed in IR were the first ones to be employed for searching the World Wide Web from 1996 to 1998. Web search, however, matured into systems that take advantage of the cross linkage available on the web, and is not a focus of the present article. In this article, I will concentrate on describing the evolution of modern textual IR systems ( [27, 33, 16] are some good IR resources).

## 2 Models and Implementation

Early IR systems were boolean systems which allowed users to specify their information need using a complex combination of boolean ANDs, ORs and NOTs. Boolean systems have several shortcomings, e.g., there is no inherent notion of document ranking, and it is very hard for a user to form a good search request. Even though boolean systems usually return matching documents in some order, e.g., ordered by date, or some other document feature, relevance ranking is often not critical in a boolean system. Even though it has been shown by the research community that boolean systems are less effective than ranked retrieval systems, many power users still use boolean systems as they feel more in control of the retrieval process. However, most everyday users of IR systems expect IR systems to do ranked retrieval. IR systems rank documents by their estimation of the usefulness of a document for a user query. Most IR systems assign a numeric score to every document and rank documents by this score. Several models have been proposed for this process. The three most used models in IR research are the vector space model, the probabilistic models, and the inference network model.

### 2.1 Vector Space Model

In the vector space model text is represented by a vector of *terms*. [28] The definition of a term is not inherent in the model, but terms are typically words and phrases. If words are chosen as terms, then every word in the vocabulary becomes an independent dimension in a very high dimensional vector space. Any text can then be represented by a vector in this high dimensional space. If a term belongs to a text, it gets a non-zero value in the text-vector along the dimension corresponding to the term. Since any text contains a limited set of terms (the vocabulary can be millions of terms), most text vectors are very sparse. Most vector based systems operate in the positive quadrant of the vector space, i.e., no term is assigned a negative value.

To assign a numeric score to a document for a query, the model measures the *similarity* between the query vector (since query is also just text and can be converted into a vector) and the document vector. The similarity between two vectors is once again not inherent in the model. Typically, the angle between two vectors is used as a measure of divergence between the vectors, and cosine of the angle is used as the numeric similarity (since

[1] `http://trec.nist.gov`

cosine has the nice property that it is 1.0 for identical vectors and 0.0 for orthogonal vectors). As an alternative, the inner-product (or dot-product) between two vectors is often used as a similarity measure. If all the vectors are forced to be unit length, then the cosine of the angle between two vectors is same as their dot-product. If $\vec{D}$ is the document vector and $\vec{Q}$ is the query vector, then the dot-product similarity between document $D$ and query $Q$ (or score of $D$ for $Q$) can be represented as:

$$Sim(\vec{D}, \vec{Q}) = \sum_{t_i \in Q,D} w_{t_{iQ}} \cdot w_{t_{iD}}$$

where $w_{t_{iQ}}$ is the value of the $i$th component in the query vector $\vec{Q}$, and $w_{t_{iD}}$ is the $i$th component in the document vector $\vec{D}$. (Since any word not present in either the query or the document has a $w_{t_{iQ}}$ or $w_{t_{iD}}$ value of 0, respectively, we can do the summation only over the terms common in the query and the document.) How we arrive at $w_{t_{iQ}}$ and $w_{t_{iD}}$ is not defined by the model, but is quite critical to the search effectiveness of an IR system. $w_{t_{iD}}$ is often referred to as the *weight* of term-$i$ in document $D$, and is discussed in detail in Section 4.1.

## 2.2 Probabilistic Models

This family of IR models is based on the general principle that documents in a collection should be ranked by decreasing probability of their relevance to a query. This is often called *the probabilistic ranking principle* (PRP). [20] Since true probabilities are not available to an IR system, probabilistic IR models *estimate* the probability of relevance of documents for a query. This estimation is the key part of the model, and this is where most probabilistic models differ from one another. The initial idea of probabilistic retrieval was proposed by Maron and Kuhns in a paper published in 1960. [18] Since then, many probabilistic models have been proposed, each based on a different probability estimation technique.

Due to space limitations, it is not possible to discuss the details of these models here. However, the following description abstracts out the common basis for these models. We denote the probability of relevance for document $D$ by $P(R|D)$. Since this ranking criteria is monotonic under log-odds transformation, we can rank documents by $log\frac{P(R|D)}{P(\bar{R}|D)}$, where $P(\bar{R}|D)$ is the probability that the document is non-relevant. This, by simple bayes transform, becomes $log\frac{P(D|R)\cdot P(R)}{P(D|\bar{R})\cdot P(\bar{R})}$. Assuming that the prior probability of relevance, i.e., $P(R)$, is independent of the document under consideration and thus is constant across documents, $P(R)$ and $P(\bar{R})$ are just scaling factors for the final document scores and can be removed from the above formulation (for ranking purposes). This further simplifies the above formulation to: $log\frac{P(D|R)}{P(D|\bar{R})}$.

Based on the assumptions behind estimation of $P(D|R)$, different probabilistic models start diverging at this point. In the simplest form of this model, we assume that terms (typically words) are mutually independent (this is often called the *independence assumption*), and $P(D|R)$ is re-written as a product of individual term probabilities, i.e., probability of presence/absence of a term in relevant/non-relevant documents:

$$P(D|R) = \prod_{t_i \in Q,D} P(t_i|R) \cdot \prod_{t_j \in Q,\bar{D}} (1 - P(t_j|R))$$

which uses probability of presence of a term $t_i$ in relevant documents for all terms that are common to the query and the document, and the probability of absence of a term $t_j$ from relevant documents for all terms that are present in the query and absent from the document. If $p_i$ denotes $P(t_i|R)$, and $q_i$ denotes $P(t_i|\bar{R})$, the ranking formula $log(\frac{P(D|R)}{P(D|\bar{R})})$ reduces to:

$$log\frac{\prod_{t_i \in Q,D} p_i \cdot \prod_{t_j \in Q,\bar{D}} (1 - p_j)}{\prod_{t_i \in Q,D} q_i \cdot \prod_{t_j \in Q,\bar{D}} (1 - q_j)}$$

For a given query, we can add to this a constant $log(\prod_{t_i \in Q} \frac{1-q_i}{1-p_i})$ to transform the ranking formula to use only the terms present in a document:

$$log \prod_{t_i \in Q,D} \frac{p_i \cdot (1-q_i)}{q_i \cdot (1-p_i)} \qquad or \qquad \sum_{t_i \in Q,D} log \frac{p_i \cdot (1-q_i)}{q_i \cdot (1-p_i)}$$

Different assumptions for estimation of $p_i$ and $q_i$ yield different document ranking functions. E.g., in [7] Croft and Harper assume that $p_i$ is the same for all query terms and $\frac{p_i}{1-p_i}$ is a constant and can be ignored for ranking purposes. They also assume that almost all documents in a collection are non-relevant to a query (which is very close to truth given that collections are large) and estimate $q_i$ by $\frac{n_i}{N}$, where $N$ is the collection size and $n_i$ is the number of documents that contain term-$i$. This yields a scoring function $\sum_{t_i \in Q,D} log \frac{N-n_i}{n_i}$ which is similar to the inverse document frequency function discussed in Section 4.1. Notice that if we think of $log \frac{p_i \cdot (1-q_i)}{q_i \cdot (1-p_i)}$ as the weight of term-$i$ in document $D$, this formulation becomes very similar to the similarity formulation in the vector space model (Section 2.1) with query terms assigned a unit weight.

## 2.3 Inference Network Model

In this model, document retrieval is modeled as an inference process in an inference network. [32] Most techniques used by IR systems can be implemented under this model. In the simplest implementation of this model, a document instantiates a term with a certain strength, and the credit from multiple terms is accumulated given a query to compute the equivalent of a numeric score for the document. From an operational perspective, the strength of instantiation of a term for a document can be considered as the *weight* of the term in the document, and document ranking in the simplest form of this model becomes similar to ranking in the vector space model and the probabilistic models described above. The strength of instantiation of a term for a document is not defined by the model, and any formulation can be used.

## 2.4 Implementation

Most operational IR systems are based on the *inverted list* data structure. This enables fast access to a list of documents that contain a term along with other information (for example, the weight of the term in each document, the relative positions of the term in each document, etc.). A typical inverted list may be stored as:

$$t_i \rightarrow < d_a, ... >, < d_b, ... >, ..., < d_n, ... >$$

which depicts that term-$i$ is contained in $d_a$, $d_b$, $\ldots$, $d_n$, and stores any other information. All models described above can be implemented using inverted lists. Inverted lists exploit the fact that given a user query, most IR systems are only interested in scoring a small number of documents that contain some query term. This allows the system to only score documents that will have a non-zero numeric score. Most systems maintain the scores in a heap (or another similar data structure) and at the end of processing return the top scoring documents for a query. Since all documents are indexed by the terms they contain, the process of generating, building, and storing document representations is called *indexing* and the resulting inverted files are called the *inverted index*.

Most IR systems use single words as terms. Words that are considered non-informative, like function words (*the, in, of, a, . . .*), also called *stop-words*, are often ignored. Conflating various forms of the same word to its root form, called *stemming* in IR jargon, is also used by many systems. The main idea behind stemming is that users searching for information on `retrieval` will also be interested in articles that have information about `retrieve, retrieved, retrieving, retriever`, and so on. This also makes the system susceptible to errors due to poor stemming. For example, a user interested in `information retrieval` might get an article titled `Information on Golden Retrievers` due to stemming. Several stemmers for various languages have been developed over the years, each with its own set of stemming rules. However, the usefulness of stemming for improved search quality has always been questioned in the research community, especially for English. The consensus is that, for English, on average stemming yields small improvements in search effectiveness; however, in cases where it causes poor retrieval, the user can be considerably annoyed. [12] Stemming is possibly more beneficial for languages with many word inflections (like German).

Some IR systems also use multi-word phrases (e.g., "information retrieval") as index terms. Since phrases are considered more meaningful than individual words, a phrase match in the document is considered more informative than single word matches. Several techniques to generate a list of phrases have been explored. These range from fully linguistic (e.g., based on parsing the sentences) to fully statistical (e.g., based on counting word cooccurrences). It is accepted in the IR research community that phrases are valuable indexing units and yield improved search effectiveness. However, the style of phrase generation used is not critical. Studies comparing linguistic phrases to statistical phrases have failed to show a difference in their retrieval performance. [8]

# 3 Evaluation

Objective evaluation of search effectiveness has been a cornerstone of IR. Progress in the field critically depends upon experimenting with new ideas and evaluating the effects of these ideas, especially given the experimental nature of the field. Since the early years, it was evident to researchers in the community that objective evaluation of search techniques would play a key role in the field. The Cranfield tests, conducted in 1960s, established the desired set of characteristics for a retrieval system. Even though there has been some debate over the years, the two desired properties that have been accepted by the research community for measurement of search effectiveness are *recall*: the proportion of relevant documents retrieved by the system; and *precision*: the proportion of retrieved documents that are relevant.[6]

It is well accepted that a good IR system should retrieve as many relevant documents as possible (i.e., have a high recall), and it should retrieve very few non-relevant documents (i.e., have high precision). Unfortunately, these two goals have proven to be quite contradictory over the years. Techniques that tend to improve recall tend to hurt precision and vice-versa. Both recall and precision are set oriented measures and have no notion of ranked retrieval. Researchers have used several variants of recall and precision to evaluate ranked retrieval. For example, if system designers feel that precision is more important to their users, they can use precision in top ten or twenty documents as the evaluation metric. On the other hand if recall is more important to users, one could measure precision at (say) 50% recall, which would indicate how many non-relevant documents a user would have to read in order to find half the relevant ones. One measure that deserves special mention is *average precision*, a single valued measure most commonly used by the IR research community to evaluate ranked retrieval. Average precision is computed by measuring precision at different recall points (say 10%, 20%, and so on) and averaging. [27]

# 4 Key Techniques

Section 2 described how different IR models can implemented using inverted lists. The most critical piece of information needed for document ranking in all models is a term's weight in a document. A large body of work has gone into proper estimation of these weights in different models. Another technique that has been shown to be effective in improving document ranking is query modification via *relevance feedback*. A state-of-the-art ranking system uses an effective weighting scheme in combination with a good query expansion technique.

## 4.1 Term Weighting

Various methods for weighting terms have been developed in the field. Weighting methods developed under the probabilistic models rely heavily upon better estimation of various probabilities. [21] Methods developed

| | |
|---|---|
| *tf* | is the term’s frequency in document |
| *qtf* | is the term’s frequency in query |
| *N* | is the total number of documents in the collection |
| *df* | is the number of documents that contain the term |
| *dl* | is the document length (in bytes), and |
| *avdl* | is the average document length |

Okapi weighting based document score: [23]

$$\sum_{t\in Q,D} ln\frac{N-df+0.5}{df+0.5}\cdot\frac{(k_1+1)tf}{(k_1(1-b)+b\frac{dl}{avdl})+tf}\cdot\frac{(k_3+1)qtf}{k_3+qtf}$$

$k_1$ (between 1.0–2.0), $b$ (usually 0.75), and $k_3$ (between 0–1000) are constants.

Pivoted normalization weighting based document score: [30]

$$\sum_{t\in Q,D}\frac{1+ln(1+ln(tf))}{(1-s)+s\frac{dl}{avdl}}\cdot qtf\cdot ln\frac{N+1}{df}$$

$s$ is a constant (usually 0.20).

Table 1: Modern Document Scoring Schemes

under the vector space model are often based on researchers’ experience with systems and large scale experimentation. [26] In both models, three main factors come into play in the final term weight formulation. a) Term Frequency (or tf): Words that repeat multiple times in a document are considered salient. Term weights based on *tf* have been used in the vector space model since the 1960s. b) Document Frequency: Words that appear in many documents are considered common and are not very indicative of document content. A weighting method based on this, called inverse document frequency (or *idf*) weighting, was proposed by Sparck-Jones early 1970s. [15] And c) Document Length: When collections have documents of varying lengths, longer documents tend to score higher since they contain more words and word repetitions. This effect is usually compensated by normalizing for document lengths in the term weighting method. Before TREC, both the vector space model and the probabilistic models developed term weighting schemes which were shown to be effective on the small test collections available then. Inception of TREC provided IR researchers with very large and varied test collections allowing rapid development of effective weighting schemes.

Soon after first TREC, researchers at Cornell University realized that using raw *tf* of terms is non-optimal, and a dampened frequency (e.g., a logarithmic *tf* function) is a better weighting metric. [4] In subsequent years, an effective term weighting scheme was developed under a probabilistic model by Steve Robertson and his team at City University, London. [22] Motivated in part by Robertson’s work, researchers at Cornell University developed better models of how document length should be factored into term weights. [29] At the end of this rapid advancement in term weighting, the field had two widely used weighting methods, one (often called *Okapi weighting*) from Robertson’s work, and the second (often called *pivoted normalization weighting*) from the work done at Cornell University. Most research groups at TREC currently use some variant of these two weightings. Many studies have used the phrase *tf-idf weighting* to refer to any term weighting method that uses *tf* and *idf*, and do not differentiate between using a simple document scoring method (like $\sum_{t\in Q,D} tf\cdot ln\frac{N}{df}$) and a state-of-the-art scoring method (like the ones shown in Table 1). Many such studies claim that their proposed methods are far superior than *tf-idf weighting*, often a wrong conclusion based on the poor weighting formulation used.

### 4.2 Query Modification

In the early years of IR, researchers realized that it was quite hard for users to formulate effective search requests. It was thought that adding synonyms of query words to the query should improve search effectiveness. Early research in IR relied on a thesaurus to find synonyms.[14] However, it is quite expensive to obtain a good general purpose thesaurus. Researchers developed techniques to automatically generate thesauri for use in query modification. Most of the automatic methods are based on analyzing word cooccurrence in the documents (which often produces a list of strongly related words). Most query augmentation techniques based on automatically generated thesaurii had very limited success in improving search effectiveness. The main reason behind this is the lack of query context in the augmentation process. Not all words related to a query word are meaningful in context of the query. E.g., even though `machine` is a very good alternative for the word `engine`, this augmentation is not meaningful if the query is `search engine`.

In 1965 Rocchio proposed using relevance feedback for query modification. [24] Relevance feedback is motivated by the fact that it is easy for users to judge some documents as relevant or non-relevant for their query. Using such relevance judgments, a system can then automatically generate a better query (e.g., by adding related new terms) for further searching. In general, the user is asked to judge the relevance of the top few documents retrieved by the system. Based on these judgments, the system modifies the query and issues the new query for finding more relevant documents from the collection. Relevance feedback has been shown to work quite effectively across test collections.

New techniques to do meaningful query expansion in absence of any user feedback were developed early 1990s. Most notable of these is *pseudo-feedback*, a variant of relevance feedback. [3] Given that the top few documents retrieved by an IR system are often on the general query topic, selecting related terms from these documents should yield useful new terms irrespective of document relevance. In pseudo-feedback the IR system assumes that the top few documents retrieved for the initial user query are “relevant”, and does relevance feedback to generate a new query. This expanded new query is then used to rank documents for presentation to the user. Pseudo feedback has been shown to be a very effective technique, especially for short user queries.

## 5 Other Techniques and Applications

Many other techniques have been developed over the years and have met with varying success. **Cluster hypothesis** states that documents that cluster together (are very similar to each other) will have a similar relevance profile for a given query. [10] Document clustering techniques were (and still are) an active area of research. Even though the usefulness of document clustering for improved search effectiveness (or efficiency) has been very limited, document clustering has allowed several developments in IR, e.g., for browsing and search interfaces. **Natural Language Processing** (NLP) has also been proposed as a tool to enhance retrieval effectiveness, but has had very limited success. [31] Even though document ranking is a critical application for IR, it is definitely not the only one. The field has developed techniques to attack many different problems like information filtering [2], topic detection and tracking (or TDT) [1], speech retrieval [13], cross-language retrieval [9], question answering [19], and many more.

## 6 Summing Up

The field of information retrieval has come a long way in the last forty years, and has enabled easier and faster information discovery. In the early years there were many doubts raised regarding the simple statistical techniques used in the field. However, for the task of finding information, these statistical techniques have indeed proven to be the most effective ones so far. Techniques developed in the field have been used in many other areas and have yielded many new technologies which are used by people on an everyday basis, e.g., web search engines, junk-email filters, news clipping services. Going forward, the field is attacking many critical problems that users face in todays information-ridden world. With exponential growth in the amount of information available, information retrieval will play an increasingly important role in future.

## References

[1] J. Allan, J. Carbonell, G. Doddington, J. Yamron, and Y. Yang. Topic detection and tracking pilot study: Final report. In *Proceedings of DARPA Broadcast News Transcription and Understanding Workshop*, pages 194–218, 1998.

[2] N. J. Belkin and W. B. Croft. Information filtering and information retrieval: Two sides of the same coin? *Communications of the ACM*, 35(12):29–38, 1992.

[3] Chris Buckley, James Allan, Gerard Salton, and Amit Singhal. Automatic query expansion using SMART: TREC 3. In *Proceedings of the Third Text REtrieval Conference (TREC-3)*, pages 69–80. NIST Special Publication 500-225, April 1995.

[4] Chris Buckley, Gerard Salton, and James Allan. Automatic retrieval with locality information using SMART. In *Proceedings of the First Text REtrieval Conference (TREC-1)*, pages 59–72. NIST Special Publication 500-207, March 1993.

[5] Vannevar Bush. As We May Think. *Atlantic Monthly*, 176:101–108, July 1945.

[6] C. W. Cleverdon. The Cranfield tests on index language devices. *Aslib Proceedings*, 19:173–192, 1967.

[7] W. B. Croft and D. J. Harper. Using probabilistic models on document retrieval without relevance information. *Journal of Documentation*, 35:285–295, 1979.

[8] J. L. Fagan. The effectiveness of a nonsyntactic approach to automatic phrase indexing for document retrieval. *Journal of the American Society for Information Science*, 40(2):115–139, 1989.

[9] G. Grefenstette, editor. *Cross-Language Information Retrieval*. Kluwer Academic Publishers, 1998.

[10] A. Griffiths, H. C. Luckhurst, and P. Willett. Using interdocument similarity in document retrieval systems. *Journal of the American Society for Information Science*, 37:3–11, 1986.

[11] D. K. Harman. Overview of the first Text REtrieval Conference (TREC-1). In *Proceedings of the First Text REtrieval Conference (TREC-1)*, pages 1–20. NIST Special Publication 500-207, March 1993.

[12] David Hull. Stemming algorithms - a case study for detailed evaluation. *Journal of the American Society for Information Science*, 47(1):70–84, 1996.

[13] G. J. F. Jones, J. T. Foote, K. Sparck Jones, and S. J. Young. Retrieving spoken documents by combining multiple index sources. In *Proceedings of ACM SIGIR'96*, pages 30–38, 1996.

[14] K. Sparck Jones. *Automatic Keyword Classification for Information Retrieval*. Butterworths, London, 1971.

[15] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[16] K. Sparck Jones and P. Willett, editors. *Readings in Information Retrieval*. Morgan Kaufmann, 1997.

[17] H. P. Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development*, 1957.

[18] M. E. Maron and J. L. Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM*, 7:216–244, 1960.

[19] Marius Pasca and Sanda Harabagiu. High performance question/answering. In *Proceedings of the 24th International Conference on Research and Development in Information Retrieval*, pages 366–374, 2001.

[20] S. E. Robertson. The probabilistic ranking principle in IR. *Journal of Documentation*, 33:294–304, 1977.

[21] S. E. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3):129–146, May-June 1976.

[22] S. E. Robertson and S. Walker. Some simple effective approximations to the 2–poisson model for probabilistic weighted retrieval. In *Proceedings of ACM SIGIR'94*, pages 232–241, 1994.

[23] S. E. Robertson, S. Walker, and M. Beaulieu. Okapi at TREC–7: automatic ad hoc, filtering, VLC and filtering tracks. In *Proceedings of the Seventh Text REtrieval Conference (TREC-7)*, pages 253–264. NIST Special Publication 500-242, July 1999.

[24] J. J. Rocchio. Relevance feedback in information retrieval. In Gerard Salton, editor, *The SMART Retrieval System—Experiments in Automatic Document Processing*, pages 313–323, Englewood Cliffs, NJ, 1971. Prentice Hall, Inc.

[25] Gerard Salton, editor. *The SMART Retrieval System—Experiments in Automatic Document Retrieval*. Prentice Hall Inc., Englewood Cliffs, NJ, 1971.

[26] Gerard Salton and Chris Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[27] Gerard Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw Hill Book Co., New York, 1983.

[28] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for information retrieval. *Communications of the ACM*, 18(11):613–620, November 1975.

[29] Amit Singhal, Chris Buckley, and Mandar Mitra. Pivoted document length normalization. In *Proceedings of ACM SIGIR'96*, pages 21–29. Association for Computing Machinery, New York, August 1996.

[30] Amit Singhal, John Choi, Donald Hindle, David Lewis, and Fernando Pereira. AT&T at TREC-7. In *Proceedings of the Seventh Text REtrieval Conference (TREC-7)*, pages 239–252. NIST Special Publication 500-242, July 1999.

[31] T. Strzalkowski, L. Guthrie, J. Karlgren, J. Leistensnider, F. Lin, J. Perez-Carballo, T. Straszheim, J. Wang, and J. Wilding. Natural language information retrieval: TREC-5 report. In *Proceedings of the Fifth Text REtrieval Conference (TREC-5)*, 1997.

[32] Howard Turtle. *Inference Networks for Document Retrieval*. Ph.D. thesis, Department of Computer Science, University of Massachusetts, Amherst, MA 01003, 1990. Available as COINS Technical Report 90-92.

[33] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

# Integrating Diverse Information Management Systems: A Brief Survey

Sriram Raghavan    Hector Garcia-Molina
Computer Science Department
Stanford University
Stanford, CA 94305, USA
{rsram, hector}@cs.stanford.edu

### Abstract

*Most current information management systems can be classified into text retrieval systems, relational/object database systems, or semistructured/XML database systems. However, in practice, many applications data sets involve a combination of free text, structured data, and semistructured data. Hence, integration of different types of information management systems has been, and continues to be, an active research topic. In this paper, we present a short survey of prior work on integrating and inter-operating between text, structured, and semistructured database systems. We classify existing literature based on the kinds of systems being integrated and the approach to integration. Based on this classification, we identify the challenges and the key themes underlying existing work in this area.*

## 1 Introduction

Most information management systems (IMS) can usually be classified into one of three categories depending on the kind of data that they are primarily designed to handle.[1] Text retrieval systems are concerned with the management and query-based retrieval of collections of unstructured text documents. Relational or object-oriented database systems are concerned with the management of structured or strictly-typed data, i.e., data that conforms to a well-defined schema. Finally, semistructured databases are designed to efficiently manage data that only partially conforms to a schema, or whose schema can evolve rapidly [1]. Each of these systems employ different physical and logical data models, query languages, and query processing techniques appropriate to the type of data being managed (see Table 1 for a brief summary).

There is a substantial body of work that deals with the design of each of these classes of information management systems. In addition, there has been significant interest in combining, integrating, and inter-operating between information management systems that belong to different classes. There are two primary motivations for most of the work in this area. First, many applications require processing of data that belongs to more than one type. For instance, a medical information system at a hospital must process doctor reports (free text documents) as well as patient records (structured relational data). Similarly, an order processing application might

**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

[1] For the purposes of this paper, we will be ignoring non-text (audio, images, and video) information management systems as well as systems that are designed for specialized data types (e.g., geographical information systems).

| | *Text Retrieval Systems* | *Relational & Object DBMS* | *Semistructured (XML) DBMS* |
|---|---|---|---|
| *Data Models* | Unstructured text documents possibly with structured fields | Relations, Objects grouped into classes | Directed edge-labeled graphs |
| *Query Models* | Extended boolean, Vector space, Probabilistic | Relational model | Relational model extended with graph operations and recursion |
| *Query Operators* | Boolean operators, Proximity operators, Pattern matching operators | Relational operators | Relational operators + (extended) path expressions + restructuring operators |
| *Example Query Languages* | Boolean, Natural language [5] | SQL, OQL | Lorel[2], UnQL [8] |

Table 1: Comparing different information management systems based on data and query models[2]

need to handle inventory information in a relational database as well as purchase orders received as (semistructured) XML documents. Second, there are significant advantages in leveraging the facilities provided by one type of information management system to implement another. For instance, a text retrieval system that is built on top of a relational or object database system [7, 38] can benefit from the sophisticated concurrency control and recovery facilities of the latter, without having to implement these features from scratch.

However, fundamental differences in the data and query models of these systems, pose significant challenges to such integration efforts. Depending on the target application scenario, there are several ways of addressing these challenges. For instance, some integration approaches are based on *extending* one type of system, either natively or through the use of plugin modules [19, 18], to support operators, features, or data types normally found in another (e.g., support for keyword searches in a semistructured database system [36]). Other approaches employ a separate *middleware* layer that provides a uniform and common query interface to a collection of diverse information management systems (e.g., the Garlic system [9] at IBM Almaden).

In this paper, we present a short survey of some of the techniques for integrating different classes of information management systems. We present a classification system that enables us to group related work, based on the types of systems being integrated and the integration architecture. We emphasize that our survey is by no means comprehensive, nor is our classification the only way to organize the literature. Our aim is to use the classification to identify key integration challenges and describe some of the proposed solutions.

The rest of the paper is organized as follows. In Section 2, we describe our classification system. In Sections 3, 4, and 5, for each of the three possible pairs of information management systems, we identify the key issues in integration and briefly discuss some representative work from the literature. We conclude in Section 6.

## 2 Classification System

We classify existing integration literature along two axes, as shown in Table 2. The horizontal axis of Table 2 enumerates all possible pairs from among the three classes of information management systems (IMS) described earlier. The vertical axis lists the three most commonly employed architectures for tying such systems together. The cells of the table are populated with bibliographic references that indicate how each of the referenced works fit into our classification system.

The schematics in Figure 1 illustrate the differences between the three integration architectures. In systems with a *layered* architecture, an IMS of one type is implemented as an application that operates over an IMS

[2] Note that the entries in this table represent only the most common cases. Individual systems might use some variations or extensions of these models and languages.

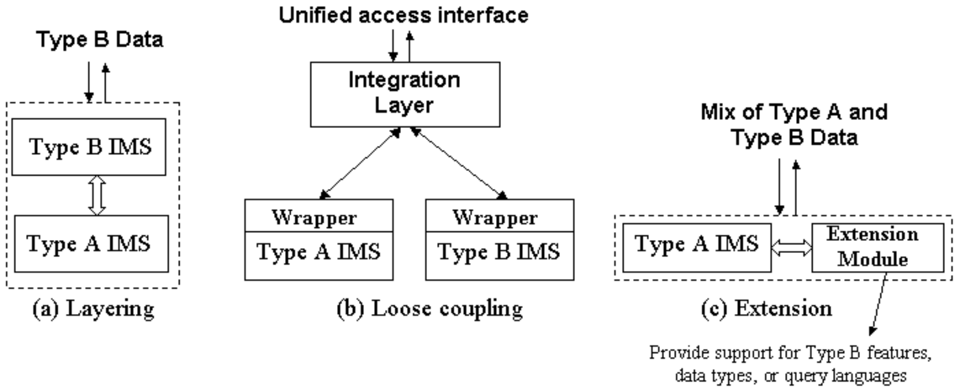


Figure 1: Common integration architectures

| | | | |
|---|---|---|---|
| *Layering* | [6][16][41][46][7][21]<br>[38][39][52][20] | [42] | [24][51][54][13]<br>[30][49][28] |
| *Loose Coupling* | [40][56][9][11][12][26][27][17] | [3] | [45][14] |
| *Extension* | [44][32][34][19][18][35][21][23]<br>[53][47][22][57][48][55][37] | [35][33][31][25]<br>[4][36] | [29][15][50][43][10] |
| | *Text Retrieval*<br>*Relational/OO* | *Text Retrieval*<br>*Semistructured* | *Relational/OO*<br>*Semistructured* |

Table 2: Classification of literature on integrating different types of information management systems

of another type. The main advantage of this approach is that the top-level IMS can leverage the facilities of the underlying IMS (e.g., concurrency control, recovery, caching, index structures, etc.), without significant additional development time and effort. However, the challenge lies in efficiently mapping the data types and operators used by the top-level IMS in terms of the types and operators supported by the underlying IMS.

*Loosely coupled* architectures isolate the integration logic in a separate integration (or mediation) layer (Figure 1(b)). This layer provides a unified access interface to the integrated system using its own data and query languages. The fundamental challenge in this architecture is to design efficient mechanisms to translate queries expressed in the unified model in terms of the query capabilities of the individual IMSs. The advantage is that unlike the other two architectures, modifications to the individual IMSs are minimal or completely unnecessary.

Finally, *extension* architectures (Figure 1(c)) enhance the capabilities of a particular type of IMS by using an extension module that provides support for new data types, operators, or query languages usually available only in IMSs of another type. When extension interfaces are available in the original IMS (as is the case with most commercial relational systems [19, 18]), the extension module can be implemented using these interfaces. Otherwise, the original IMS is modified to natively support the new features.

Note that even though we have explicitly distinguished between these three architectures to help navigate the literature, actual implementations sometimes have flavors of more than one architecture. For instance, reference [21] proposes *extensions* to a relational DBMS to facilitate efficient implementations of inverted indexes and then *layers* a text retrieval system atop the enhanced RDBMS. Similarly, for efficiency reasons, some systems push the integration layer of the *loosely coupled* architecture into one of the individual systems [37].

# 3 Text Retrieval and Relational/Object Database Systems

The integration of information retrieval (IR) and traditional database systems has long been recognized as a challenging and difficult task. Fundamental differences in the query and retrieval models (precisely defined declarative queries and exact answers in databases versus imprecise queries and approximate retrieval in IR systems) have resulted in vastly different query languages, index structures, storage formats, and query processing techniques. Both the IR and DB communities have attempted to address this problem, but with different goals, and by adopting different architectures. The database community has favored the *extension* architecture with the aim of efficiently providing IR features within the DBMS framework (lower left cell of Table 2). In contrast, the IR community has shown a preference for the *layered* architecture, with the aim of exploiting DBMS features (concurrency control, recovery, security, transaction semantics, robustness, etc.) to build more scalable and robust text retrieval systems (top left cell of Table 2). Though less popular, there has also been some work in building loosely coupled IR-DB systems [17]. In the interest of space, we do not discuss such loosely-coupled systems in this section.

## 3.1 Extensions to Database Systems

The extension-based approach of the DB community has led to work on extended relational models and algebras, extensions to database query languages, new index structures [44] and data types [53], and query execution strategies for optimizing IR-style text operations.

Extensions to the relational model fall into two categories - nested (non-first normal form or $NF^2$) relational models [22, 48, 47, 57] to capture hierarchical document structure and probabilistic models [32, 34] to incorporate uncertainty and imprecision into the DBMS framework. Such probabilistic extensions, though promising, require substantial changes to the core query processing algorithms of database systems and as a result, are not yet a part of actual implementations.

Reference [21] proposes the technique of cooperative indexing, where the database is responsible for inverted index storage and access but the IR extension defines the actual contents of the index. References [37] and [23] address performance issues in implementing database extensions - specifically, extensions to integrate with external (i.e., outside the database system) text retrieval systems. The SQL/MM Full-Text [53] standard attempts to standardize the integration of text retrieval with SQL database systems by providing definitions for text-related abstract data types. Finally, in [35], Goldman et al. propose an extension to relational and object databases by introducing a proximity operator, called the NEAR operator, adapted from the notion of textual proximity used in text retrieval systems.

## 3.2 Layering IR Systems atop Relational/Object Database Systems

Some of the earliest attempts at integrating IR and DB systems treated a text retrieval system as a database application that was implemented on top of a standard relational DBMS [6, 16, 41, 46, 39, 38]. The inverted index, the lexicon, and other term frequency statistics were stored in standard database tables. IR queries were translated into SQL queries over these tables and executed by the database. In addition, several prototype text retrieval systems have also been built using object database systems [7, 20, 52]. Since the object data model natively supports nesting, in addition to collection types and sets, we expect that systems for content-based retrieval of structured documents could be effectively implemented on top of OODB systems.

# 4 Text Retrieval and Semistructured Database Systems

XML has emerged as the de facto standard for representing semistructured information and for exchanging structured data between applications. Since XML shares the same graph-based data model as several other semistructured database query languages [2, 8], to date, most of the work on querying, indexing, and searching XML corpora has its origins in the database community (see Section 5). In [33], Fuhr and Grossjohann refer to this approach as the *data-centric* view of XML.

However, the alternative *document-centric* [33] view has only recently received the attention of the research community. This approach treats an XML corpus as a collection of logically structured text documents. By extending IR models and indexes to encode the structure and semantics of XML documents, it becomes possible to apply well-known IR techniques and support keyword searches, similarity-based retrieval, automatic classification, and clustering, of XML corpora.

In [31], the authors describe an approach for integrating keyword searches with XML query processing. They extend the XML-QL query language by introducing a new "contains" predicate for keyword-based search operations. They define the precise semantics of the extended query language and describe how to efficiently execute queries that involve keyword search as well as non-text operations. In the same vein, reference [33] describes XIRQL, an extension to the XQL query language to support IR-related features such as weighting, ranking, relevance-oriented search, and vague predicates. The Xyleme warehousing system [4] supports text search and pattern matching operations over XML documents.

As is evident from the descriptions in the previous paragraph, much of the work in XML-IR integration is based on the *extension* approach (Figure 1(c)), adding IR-style features to XML databases and query languages. As an example of a *layered* system, in [42], Hayashi et al. describe their implementation of a relevance ranking based XML search engine on top of a standard text retrieval system. Finally, in [3], Adar describes a personal information management system that is implemented by loosely coupling the Lore semistructured database system with the Haystack personal information retrieval system.

# 5 Relational/Object and Semistructured Database Systems

With the advent of XML as the dominant standard for information interchange, the integration of XML with relational and object database systems is an extremely active research area. Techniques for mapping the XML (semistructured) data model to the relational or object data model, for exporting relational data as XML documents, for providing XML views of relational data, and for extending relational query engines to process queries over XML data, are topics currently being investigated by the research community.

In the commercial arena, most major relational database vendors already provide support for an XML data type to natively store and manage XML documents, as well as some primitive programming APIs for importing and exporting XML documents to and from database tables [15].

In the interest of space, for the rest of the section, we shall discuss only extension and layered approaches to integrating relational/object and semistructured database systems. However, there has been some work in loosely coupling such systems [45, 14], motivated mainly by the use of XML as the integration language.

## 5.1 Extensions to Relational/Object Database Systems

Since XML is intended as a language for inter-enterprise information interchange, it is natural that techniques for publishing relational data as XML documents are in great demand. Several commercial tools already provide this functionality, but with some limitations. For instance, Oracle's XSQL [15] tool generates a fixed canonical mapping of the relational data into XML documents, by mapping each relation and attribute to an XML element, and nesting tuple elements within table elements. However, it is incapable of mapping to arbitrary XML DTDs. IBM's DB2 XML Extender supports a language for composing relational data into arbitrary XML as well as to decompose XML documents into relations.

In general, there are two parts to designing a system for publishing relations as XML documents. The first is the need for a language to specify the conversion/mapping from relations to XML documents. The second is an efficient implementation strategy to actually carry out the conversion. The SilkRoute system described in [29] was one of the earliest research prototypes that supported automatic XML generation from relational tables. SilkRoute used a language called RXL, based on a combination of SQL and XML query languages, for specifying mappings of relational tables to arbitrary XML DTDs. Using this language, it is possible to define XML views of the relational data. SilkRoute efficiently executes queries over these XML views by materializing only the portion of the XML that is required to answer the query. In [50], Shanmugasundaram et al. propose a simple language based on SQL with minor extensions for specifying the mappings. They compare different implementation alternatives and report significant performance gains from constructing XML documents as much as possible inside the relational engine.

In [43], the authors describe Ozone, an extension of an object database system to handle both structured and semistructured data. The authors extend the standard ODMG object model and the OQL query language, to handle semistructured data based on the OEM data model and Lorel query language [2].

### 5.2 Layering Semistructured Database Systems atop Relational/Object Database Systems

To design a database-backed XML repository, one must precisely define (i) a mapping from XML documents to tables or objects, (ii) an algorithm for translating queries over XML documents into SQL or OQL queries over the underlying database, and (iii) a mechanism for translating the result of database query execution into XML. There are several proposals for implementing XML repositories on top of relational [24, 30, 49, 51] and object [54, 13, 28] database systems, differing in their choices for (i),(ii), and (iii).

There are three basic alternatives for mapping XML documents into relational tables. The simplest, and least useful mapping, is to store an entire XML document as a *single database attribute*. Another possibility is to interpret XML documents as *graph structures* and supply a relational schema that can store such graphs [30, 49]. A third approach is to map the *structure* of the XML documents, (e.g., expressed as a DTD) into a corresponding relational schema and to store the documents based on these mappings [51, 24]. Only the last approach allows the repository to fully exploit the query processing and optimization capabilities of the underlying database system.

The STORED system described in [24] uses data mining to separate XML documents into structured and semistructured components. The structured component is stored in a relational database and the semistructured component is stored in a separate overflow semistructured database. In contrast, Shanmugasundaram et al. [51] store the repository entirely within the relational database. They assume that the input XML has an associated DTD and also impose restrictions on the set of DTDs that they can handle.

Techniques for mapping XML documents into object databases tend to be considerably simpler, because the object model naturally supports a hierarchical structure, collection types, and structured types such as sets and lists. Usually, a straightforward analysis of the DTD can be used to generate object type definitions (for example, in ODL) and IDREFs and IDs in the XML document can be mapped to object references and object IDs in the database system [54, 13]. However, there are two key challenges that need to be addressed. First is the fact that OODB systems are generally strongly typed whereas XML, being semistructured, is not. As a result, most often, the object model of the database must be extended, before it can be used for implementing the XML repository. Second, many OODB systems support only simple path expressions whereas most XML query languages include regular path expressions. References [54, 13, 28] address some of these challenges.

## 6 Conclusions

The design of integrated information management systems that can seamlessly handle unstructured, structured, and semistructured data, is a topic with significant research and commercial interest. In this paper, we presented a short survey of prior work on designing such integrated systems.

A visual inspection of Table 2 allows us to draw some conclusions about major research themes and focus areas. For instance, it is clear that so far, a significant portion of work on integrated information systems has involved layering other systems atop relational/object databases and on extending the capabilities of the latter (four corner cells of Table 2). However, we expect that with recent interest in middleware integration, loosely coupled systems will receive a lot of research attention in the future. Also, as mentioned in Section 4, integration of semistructured data in general, and XML in particular, with the relational/object database world has received a lot more attention than corresponding integration with text retrieval systems (comparing two rightmost columns of Table 2). We believe that this will change, once efforts to incorporate IR-style operators and structured text retrieval models into XML query languages, bear fruit.

In this paper, we have concentrated mainly on integration of pairs of systems. However, the Web provides us with a large and interesting data set that shares characteristics with all three types of data that we have dealt with in this paper. For instance, a repository of Web pages could be treated as a large collection of text documents. It could also be treated as a huge graph database since Web pages link to each other through hypertext links. Finally, each page in a Web repository can be associated with simple attributes (e.g., URL, page length, domain name, crawl date, etc.) that are easily managed in a relational database. Hence, complex queries over such repositories are likely to involve search terms connected by Boolean operators, navigational operators to refer to pages based on their interconnections, as well as predicates on page attributes. We believe that a query processing architecture that can efficiently execute such queries over huge Web repositories is an exciting research topic with applications in Web search and mining.

## References

[1] S. Abiteboul. Querying semi-structured data. In *Proc. of the 6th Intl. Conf. on Database Theory*, pages 1–18, January 1997.

[2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *Intl. Journal on Digital Libraries*, 1(1):68–88, April 1997.

[3] E. Adar. Hybrid-search and storage of semi-structured information. Master's thesis, MIT, May 1998.

[4] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML documents in Xyleme. In *Proc. of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, July 2000.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, New York, 1999.

[6] D. C. Blair. An extended relational document retrieval model. *Information Processing and Management*, 24(3):349–371, 1988.

[7] E. W. Brown, J. P. Callan, W. B. Croft, and J. E. B. Moss. Supporting full-text information retrieval with a persistent object store. In *4th Intl. Conf. on Extending Database Technology*, pages 365–378, March 1994.

[8] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 505–516, June 1996.

[9] M. J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. of the 5th Intl. Workshop on Research Issues in Data Engineering - Distributed Object Management*, pages 124–131, 1995.

[10] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *Proc. of 26th Intl. Conf. on Very Large Data Bases*, pages 646–648, September 2000.

[11] S. Chaudhuri, U. Dayal, and T. W. Yan. Join queries with external text sources: execution and optimization techniques. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 410–422, May 1995.

[12] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 91–102, June 1996.

[13] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 313–324, May 1994.

[14] V. Christophides, S. Cluet, and J. Simèon. On wrapping query languages and efficient XML integration. In *Proc. of 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 141–152, May 2000.

[15] Oracle Corporation. XML support in Oracle 8 and beyond. http://www.oracle.com/xml/.

[16] W. B. Croft and T. J. Parenty. A comparison of a network structure and a database system used for information retrieval. *Information Systems*, 10(4):377–390, 1985.

[17] 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 Intl. ACM SIGIR Conf.*, pages 223–232, June 1992.

[18] IBM Informix DataBlade. http://www-4.ibm.com/software/data/informix/blades/, 2000.

[19] DB2 Universal Database Extenders. http://www-4.ibm.com/software/data/db2/extenders/, 2000.

[20] A. P. de Vries and A. N. Wilschut. On the integration of IR and databases. In *Proc. of the 8th IFIP 2.6 Working Conf. on Database Semantics*, January 1999.

[21] S. DeFazio, A. M. Daoud, L. A. Smith, J. Srinivasan, W. B. Croft, and J. P. Callan. Integrating IR and RDBMS using cooperative indexing. In *Proc. of the 18th Intl. ACM SIGIR Conf.*, pages 84–92, July 1995.

[22] B. Desai, P. Goyal, and F. Sadri. Non first normal form universal relations: an application to information retrieval systems. *Information Systems*, 12(1):49–55, 1987.

[23] S. Deßloch and N. M. Mattos. Integrating SQL databases with content-specific search engines. In *Proc. of 23rd Intl. Conf. on Very Large Data Bases*, pages 528–537, August 1997.

[24] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pages 431–442, June 1999.

[25] D. Egnor and R. Lord. Structured information retrieval using XML. In *Proc. of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, July 2000.

[26] R. Fagin. Fuzzy queries in multimedia database systems. In *Proc. of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, June 1998.

[27] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 2001.

[28] L. Fegaras and R. Elmasri. Query engines for Web-accessible XML data. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pages 251–260, September 2001.

[29] M. Fernandez, W.-C. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. In *Proc. of the 9th Intl. World Wide Web Conf.*, pages 723–745, May 2000.

[30] D. Florescu and D. Kossman. Storing and querying XML data using a RDBMS. *Bulletin of the Technical Committee on Data Engineering*, 22(3), 1999.

[31] D. Florescu, I. Manolescu, and D. Kossmann. Integrating keyword search into XML query processing. In *Proc. of the 9th Intl. World Wide Web Conf.*, pages 119–136, May 2000.

[32] N. Fuhr. A probabilistic framework for vague queries and imprecise information in databases. In *Proc. of the 16th Intl. Conf. on Very Large Data Bases*, pages 696–707, August 1990.

[33] N. Fuhr and K. Grossjohann. XIRQL: A query language for information retrieval in XML documents. In *Proc. of the 24th ACM SIGIR Conf.*, pages 172–180, September 2001.

[34] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 15(1):32–66, 1997.

[35] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proc. of 24th Intl. Conf. on Very Large Data Bases*, pages 26–37, August 1998.

[36] R. Goldman and J. Widom. Interactive query and search in semistructured databases. In *Proc. of the First Intl. Workshop on the Web and Databases (WebDB)*, pages 42–48, March 1998.

[37] R. Goldman and J. Widom. WSQ/DSQ: A practical approach for combined querying of databases and the web. In *Proc. of 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 285–296, May 2000.

[38] D. A. Grossman and J. R. Driscoll. Structuring text within a relation system. In *Proc. of the 3rd Intl. Conf. on Database and Expert System Applications*, pages 72–77, September 1992.

[39] D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating structured data and text: A relational approach. *Journal of the American Society for Information Sciences*, 48(2):122–132, 1997.

[40] J. Gu, U. Thiel, and J. Zhao. Efficient retrieval of complex objects: Query processing in a hybrid DB and IR system. In *Proc. of the 1st German National Conf. on Information Retrieval*, 1993.

[41] D. J. Harper and A. D. M. Walker. ECLAIR: An extensible class library for information retrieval. *Computer Journal*, 35(3):256–267, 1992.

[42] Y. Hayashi, J. Tomita, and G. Kikui. Searching text-rich XML documents with relevance ranking. In *Proc. of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, July 2000.

[43] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. In *Research Issues in Structured and Semistructured Database Programming, 7th Intl. Workshop on Database Programming Languages*, September 2000.

[44] C. A. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *Proc. of the Fourteenth Intl. Conf. on Very Large Data Bases*, pages 306–317, August 1988.

[45] I. Manolescu, D. Florescu, and D. Kossman. Answering XML queries on heterogeneous data sources. In *Proc. of the 27th Intl. Conf. on Very Large Data Bases*, pages 241–250, September 2001.

[46] I. A. McLeod and R. G. Crawford. Document retrieval as a database application. *Information Technology: Research and Development*, 2:43–60, 1983.

[47] R. Sacks-Davis, A. J. Kent, K. Ramamohanarao, J. A. Thom, and J. Zobel. Atlas: A nested relational database system for text applications. *Transactions on Knowledge and Data Engineering*, 7(3):454–470, 1995.

[48] H.-J. Schek and P. Pistor. Data structures for an integrated data base management and information retrieval system. In *Proc. of the 8th Intl. Conf. on Very Large Data Bases*, pages 197–207, September 1982.

[49] A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *WebDB (Informal Proceedings)*, pages 47–52, 2000.

[50] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proc. of 26th Intl. Conf. on Very Large Data Bases*, pages 65–76, September 2000.

[51] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of 25th Intl. Conf. on Very Large Data Bases*, pages 302–314, September 1999.

[52] G. Sonnenberger. Exploiting the functionality of object-oriented database management systems for information retrieval. *Bulletin of the Technical Committee on Data Engineering*, 19(1):14–23, 1996.

[53] SQL/MM Full-text - ISO/IEC 13249. http://www.wiscorp.com/sql/sqlfulltext.zip.

[54] R. van Zwol, P. M. G. Apers, and A. N. Wilschut. Modeling and querying semistructured data with MOA. In *Proc. of the Workshop on Semistructured Data and Non-Standard Data Types*, 1999.

[55] S. R. Vasanthkumar, J. P. Callan, and W. B. Croft. Integrating INQUERY with an RDBMS to support text retrieval. *Bulletin of the Technical Committee on Data Engineering*, 19(1):24–33, 1996.

[56] M. Volz, K. Aberer, and K. Bohm. An OODBMS-IR coupling for structured documents. *Bulletin of the Technical Committee on Data Engineering*, 19(1):34–42, 1996.

[57] J. Zobel, J. A. Thom, and R. Sacks-Davis. Efficiency of nested relational document database systems. In *Proc. of the 17th Intl. Conf. on Very Large Data Bases*, pages 91–102, September 1991.

# CALL FOR PAPERS

**Kowloon Shangri-La Hotel**

**Hong Kong SAR, China**

**August 20-23, 2002**

**http://www.cs.ust.hk/vldb2002**

VLDB 2002 will continue the 27-year tradition of VLDB conferences as the premier international forum for database researchers, vendors, practitioners, application developers, and users. We invite submissions reporting original results on all technical aspects of data management as well as proposals for panels, tutorials, demonstrations, and exhibits that will present the most critical issues and views on practical leading-edge database technology, applications, and techniques. We also invite proposals for events and workshops to take place at the Conference site before or after VLDB 2002. More details about the topics of interest and the submission guidelines can be found on the conference web site.

## Important Dates

| | |
|---|---|
| **12 February 2002** | Abstract Submission Deadline |
| **19 February 2002** | Paper, Panel Demonstration and Tutorial Submission Deadline |
| **6 May 2002** | Notification of Acceptance |
| **14 June 2002** | Camera Ready Papers Due |
| **20 August 2002** | Conference Opens |

**Conference Co-chairs**
Frederick H.Lochovsky, HKUST, Hong Kong SAR
Wang Shan, People's U. China, P.R. China

**Technical Program Chair**
Philip A. Bernstein, Microsoft Corp., U.S.A.

**Local Organization Co-chairs**
Qing Li, City U. Hong Kong, Hong Kong SAR
Jianzhong Li, Harbin Inst. Technology, P.R. China

**Publicity and Publications Chair**
Dimitris Papadias, HKUST, Hong Kong SAR

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903