Optimizing Queries on Files

Mariano P. Consens
University of Waterloo
Waterloo, Canada N2L 3G1
mconsens@uwaterloo.ca

Tova Milo
CSRI, University of Toronto
Toronto, Canada M5S 1A1
milo@db.toronto.edu

Abstract

We present a framework which allows the user to access and manipulate data uniformly, regardless of whether it resides in a database or in the file system (or in both). A key issue is the performance of the system. We show that text indexing, combined with newly developed optimization techniques, can be used to provide an efficient high level interface to information stored in files. Furthermore, using these techniques, some queries can be evaluated significantly faster than in standard database implementations. We also study the tradeoff between efficiency and the amount of indexing.

1 Introduction

Database systems provide powerful features for manipulating large quantities of data. They do not, however, provide access to data stored in files outside the database system. A large portion of the information in a computerized environment resides in the file system, and not in databases. This includes, in particular, semi-structured textual information such as: electronic documents, programs, log files, online newspapers, patent information, literature citations, business profiles, and e-mail. The tools available for manipulating such files do not provide high level query and update facilities as exist in database systems. The purpose of this research is to bridge the gap between databases and the surrounding environment, and to provide a uniform framework where data can be accessed and manipulated, regardless of whether it resides in a database or in the file system (or in both). Similar motivations are presented in [SLS+93, ACM93, Sch93, GNOT92, BGMM93, Pae93, BGH+92].

In order to allow data stored in files to benefit from standard database technology, and in particular be queried and updated using database languages, we need to address two issues. First, we need to define a mapping between files and databases. This mapping should allow the effective translation of queries and updates on databases to operations on files, and vice versa. An example for such a mapping is given in [ACM93]. Second, we want to use this mapping to provide an efficient high level database interface to files.

A key issue is the performance of the system. To have feasible execution of queries and updates on files, the query evaluation and optimization mechanisms used in standard databases must be extended. In this paper we concentrate on query evaluation. To answer queries on files, one would like to avoid scanning the whole file system. We show in the paper that this can be achieved using advanced text indexing techniques. The idea is to translate high level queries on files into low level expressions that manipulate indices, and then evaluate these expressions efficiently using the engine of an indexing system. We do not address the issues of building and maintaining the indices, we assume that this is a service given by the underlying text indexing system. In this work we use the PAT[1] text indexing system [Ope93], and translate database queries on files to expressions in the PAT algebra [Gon87, ST92]. Specifically, we show how *word* indexing and *region* indexing can be combined with extended database query optimization to provide efficient access to files. Using these techniques, some queries can be evaluated significantly faster than in standard database implementations.

Database queries may be expressed by several index expressions, each affording different degree of efficiency. Clearly, we want to evaluate a query using the most efficient expression. We therefore study optimization of index expressions, and present a poly-

An earlier version was published in Proceedings of the 1994 ACM SIGMOD Conference.

[1] PAT is a registered trademark of Open Text Corporation.

nomial optimization algorithm that finds the most efficient expression that is equivalent to a given one.

As in standard database systems, it turns out that there is a tradeoff between efficiency and the amount of data being indexed. We study how the selection of specific indices affects the amount of data needed to be actually scanned when answering a query. We also present guidelines for profitable choice of indices. We show that there are cases where database queries can be fully computed using the indexing engine. In other cases, the indices are not sufficient for full computation, but can be used to locate file regions that are potentially relevant for the query computation, and thus save in file scanning. The potentially relevant regions must then be further processed, in which case we describe how extended database optimizations techniques can be used to improve the performance.

The ideas presented in this paper can be used to build a unified and efficient interface to databases and file systems. The main advantage is that the system can be implemented on top of any suitable DBMS, file system, and indexing system, and can use any standard parser for the mapping between the file and the database.

In particular, the Hy^+ System [CM93] is an example of a system that demonstrates how deductive database technology can be combined with PAT to evaluate a mixture of traditional and textual queries expressed in the visual language GraphLog [Con89, CM90]. One of the specific applications of Hy^+ where textual queries were certainly useful was the querying and visualization of software engineering data. A discussion of the implementation approach taken to integrate PAT into Hy^+ can be found in [Yeu93]. In addition, the optimization techniques presented here were tested on a prototype system combining PAT with the O_2 database system [BCD89], and the Yacc parser [AJ74].

In Section 2 we present an example used in the rest of the paper. Section 3 presents the indexing engine, and the main properties of the index algebra used in the optimization process. Section 4 introduces *structuring schemas* – a tool for specifying how a file should be interpreted in a database. The following sections describe the optimization process. Section 5 considers full indexing. Section 6 considers partial indexing, and Section 7 provides guidelines for profitable choice of indices. Finally, we conclude in Section 8.

```
@INCOLLECTION{ Corl82a,
 AUTHOR    = "G. F. Corliss and Y. F. Chang",
 TITLE     = "Solving Ordinary Differential
              Equations Using Taylor Series",
 BOOKTITLE = "Automatic Differentiation Algorithms",
 YEAR      = "1982",
 EDITOR    = "A. Griewank and G. F. Corliss",
 PUBLISHER = "SIAM",
 ADDRESS   = "Philadelphia, Penn.",
 PAGES     = "114--144",
 REFERRED  = "[Aber88a]; [Corl88a]; [Gupt85a].",
 KEYWORDS  = "point algorithm; Taylor series;
              radius of convergence;",
 ABSTRACT  = "A Fortran pre-processor uses automatic
              differentiation to write a Fortran
              program to solve the system."}
```

Figure 1: A sample bibliographic reference in BibTeX.

2 Example

We are interested in files that have strong inner structure (e.g., electronic documents, programs, SGML files). Our goal is to use the inner structure of files for providing high level and efficient interface to the information they contain. Bibliography files constitute an example of semi-structured data with which all researchers are well acquainted. The text in Figure 1 describes one bibliographic entry in the familiar BibTeX format [Lam85].

There is a multitude of bibliographic files that are available on the Internet. Even at the local level, it is common that each one of the members of a research group keeps several such files on a variety of subjects, and all of the members may share access to those bibliographies. In this familiar scenario, there is a strong motivation for being able to express queries against the information in those files.

It is easy to see that bibliography files have a well defined complex structure. We can find (i) atomic fields like `YEAR`, (ii) set valued fields like `KEYWORDS`, (iii) nested structures like `AUTHOR` and `EDITOR` that are sets of names of people, where each name consists of a first and last name, (iv) references to other entries like each of the elements in the set valued field `REFERRED`, and (v) chunks of unstructured text, both long (e.g., `ABSTRACT`) and short (e.g., `TITLE` and `BOOKTITLE`).

Viewing this information as a database provides both modeling and processing benefits. We assume in the following basic knowledge about object oriented databases. We use below the data model and query language of XSQL [KKS92]. With suitable variations we could have used any other object oriented data model and language. A bibliography database may contain classes like: *References*, *Authors*, and *Editors*. Every reference object may have atomic attributes like *Year*, and set attributes like *Authors*, (where each *Author* has a *First_Name* and *Last_Name* attributes). This bibliography database can support complex queries like "*Select the names of editors who never wrote a paper with any of the keywords occurring in a book that they edited*".

Note that this query can be easily formulated in XSQL, but cannot be directly expressed by standard text search tools (e.g. grep) that work directly on the file, and not even by most text retrieval systems, since those systems do not support join-like operations. Our research, thus, has two goals: (i) enabling users to view files as databases, and (ii) using this database view for manipulating the files.

It was shown in [ACM93] that *structuring schemas* can be used to specify how data stored in a file should be interpreted in a database. The main question is how to avoid scanning the whole file system when answering queries over such a database view. We show below that this unnecessary effort can be avoided if some information about the file content and structure is indexed and maintained by the system. We start by giving an intuition of the process with a very simple query. We then present in the next sections the full optimization algorithm.

Suppose that we want to find references where "Chang" is one of the authors. This can be formulated by an SQL query

$$Q = \texttt{SELECT}\ r\ \texttt{FROM}\ References\ r$$
$$\texttt{WHERE}\ r.Authors.Name.Last\_Name = \text{"}Chang\text{"}$$

Assume that we pre-process the file, and build two kinds of indices. *Word* index, recording the location(s) of all the words in the file, and *region* indices recording the location of various regions in the file. Assume that we built 3 region indices. The first records the locations of *Reference* regions (i.e. where each reference in the file starts, and where it ends). The second records the locations of regions corresponding to *Authors* (i.e regions starting with `AUTHOR=` and ending with a comma). The third index records locations of *Last-Names*.

This pre-processed information can be used to locate the references required by the query without actually accessing the BibTeX file. The references required by the query are exactly those corresponding to *Reference* regions, that include some *Authors* region, that includes a *Last-Name* region, that contains the word "Chang". Note that the PAT text indexing systems has facilities for indexing words and regions, and it provides a very efficient inclusion test. Thus, using a text indexing system for the pre-processing stage, and for the computation, results in a significant performance advantage.

In the above example all the necessary information (i.e. words, references, authors, and last-names) was indexed in advance. In general we would like to index as little as possible, to save in space and update cost. Assume, for example, that we decide not to index the *authors* regions. Note that a reference region contains two kinds of last names. Last names of authors, and last names of editors. If the authors regions are not indexed, then it is impossible to distinguish between references where Chang is an author and those where Chang is only an editor, without actually accessing the file. Thus the query cannot be fully computed using the indexing engine.

However, the indexed information can be still used to improve performance. In particular, it can be used locate the references that are "potentially relevant" for the query computation, and thus save on file scanning. The *Reference* regions that include some *Last-Name* region that is the word "Chang", are a superset of the required references (in those references, Chang is either an author or an editor). The number of these potentially relevant references is significantly smaller than the number of all the references in the file system. Thus scanning those references, (in order to filter out the irrelevant references), instead of scanning the whole file system, provides big performance improvement. Furthermore, we show in the paper that these potential regions can be processed efficiently using various optimizations techniques.

Another way to reduce the amount of indexed data is to use selective indexing. Assume that users often query names of authors, but never (or hardly ever) query names of editors. In that case, instead of indexing all the last-name regions it is better to index only last names of authors. Selective indexing can also be done for words. As in standard database systems, we show that there is a tradeoff between efficiency and the amount of data being indexed.

In the next sections we explain how queries on the database view of files are transformed to expressions manipulating word and region indices. Then we show how these expressions can be optimized. We consider full and partial indexing and the performance acceleration gained by them.

3 Indexing and Optimization

In this section we provide a brief overview of the PAT algebra [ST92], the language used by the PAT text retrieval system [Ope93]. PAT combines traditional text search capabilities (lexical, proximity, contextual, boolean, see [SM83]) with some original powerful features (position and frequency search). In particular, we are interested in PAT's ability to manipulate *regions* of text (in the sense that we discussed informally in the previous section). Regions are a generalization of the concepts of *document* and *field* usually found in conventional information retrieval systems. Similar modelling capabilities are described in [Bur92].

The first subsection presents a subset of the PAT algebra (with some extensions) that deals with regions. In the second part of this section we deal with a powerful optimization technique for expressions in this algebra.

3.1 The Region Algebra

PAT is a set-at-a-time algebra for text queries. There are two types of sets in the algebra: sets of match points (specific positions in the text) and sets of regions. The match points correspond to the position in the text of indexed strings (the entries of the word index referred to earlier). Each region is a substring of the indexed text, and is defined by a pair of positions in the text corresponding to the beginning and end of the region. We use the notation $r \supset s$, where r,s are two regions, to denote the fact that the region r includes the region s (i.e., the endpoints of s are within those of r).

To simplify the presentation, (and highlight the aspects of the PAT algebra that are of interest to us), we describe below a subset of the algebra (with some extentions), that concentrates on the manipulation of sets of regions. We call this algebra the **region algebra**. In particular, we assume that we are given a specific set of named regions on the indexed text[2].

A **region index** $\mathcal{I}$ is a set of region names $R_1, \ldots, R_n$. An **instance** of a region name R_i is a set of regions in a file (with no restrictions on overlaps). An **instance** I of a region index $\mathcal{I}$ is a mapping associating an instance $R_i(I)$ to each region name R_i. As a notational convenience when I is understood from the context, we use R_i for both the region name and the instance $R_i(I)$. **Region expressions** over $\mathcal{I}$ are expressions generated by the grammar

$$\begin{array}{rl} e \rightarrow & R_i \mid e \cup e \mid e \cap e \mid e - e \mid \sigma_w(e) \mid \iota(e) \mid \omega(e) \mid \\ & e \supset e \mid e \subset e \mid e \supset_d e \mid e \subset_d e \mid (e) \end{array}$$

where the terminals R_i are the region names in $\mathcal{I}$.

[2]Note that the full PAT algebra is capable of constructing sets of regions dynamically. From the point of view of this work we can treat regions defined dynamically as if they were *views*.

Given a region expression e and an instance I , $e(I)$ denotes the result of evaluating e on I, where the semantics are as follows. The operations $\cup$ (union), $\cap$ (intersection), and $-$ (difference), are the usual set theoretic operations on sets of regions. The *selection* operation σ_w takes a set of regions R and returns the regions $r \in R$ containing (exactly) the word w. (The selection is implemented by combined usage of the word and region indices). The *innermost* ι (resp. *outermost* ω) operation takes a set of regions R and returns the $r \in R$ such that there is no $r' \in R, r' \neq r$ for which $r \supset r'$ (resp. $r' \supset r$).

The $\supset$ (*including*) and $\subset$ (*included*) operations take two sets of regions R and S and return the sets of regions

$$R \supset S = \{r \in R : \exists s \in S, r \supset s\}$$

$$R \subset S = \{r \in R : \exists s \in S, s \supset r\}$$

Finally, the $\supset_d$ (*directly including*) and $\subset_d$ (*directly included*) operations are a refinement of $\supset$ and $\subset$ resp. The operation $\supset_d$ ($\subset_d$) selects regions $r \in R$ that directly include (are directly included in) a region $s \in S$, i.e. there is no other indexed region between r and s. More formally,

$$\begin{array}{rl} R \supset_d S = & \{r \in R : \exists s \in S, r \supset s \wedge \\ & \neg \exists t \in T, T \in \mathcal{I}, r \supset t \supset s\} \\ R \subset_d S = & \{r \in R : \exists s \in S, s \supset r \wedge \\ & \neg \exists t \in T, T \in \mathcal{I}, s \supset t \supset r\} \end{array}$$

We show below how $\supset_d$ can be computed using the other algebra operators, by an algorithm that additionally uses a *while* construct ($\subset_d$ can be computed similarly). The main objective of this presentation is to give intuition about the cost of this operation, and in particular to show that it is significantly more expensive than the simple inclusion operation $\supset$.

The program, which takes as input two regions R, S and produces as output $R_{result} = R \supset_d S$, basically iterates over nested layers of R regions, and for each layer selects the R regions of the layer that directly include an S region.

$R_{layer} := \omega(R)$; $R_{rest} := R - R_{layer}$; $R_{result} := \emptyset$;
while $(R_{layer} \supset S) \neq \emptyset$ **do**
  $R_{result} := R_{result} \cup$
    $(R_{layer} \supset (S - (\bigcup_{T \in \mathcal{I} - \{S\}} (S \subset T \subset R_{layer}))))$;
  $R_{layer} := \omega(R_{rest})$; $R_{rest} := R_{rest} - R_{layer}$;
end
return R_{result}

Note that $\supset, \supset_d, \subset, \subset_d$ are not associative. For brevity, we omit parentheses and assume that the operations are grouped from the right. The following is an example for a region expression. Consider the region index $\mathcal{I} = \{$ *Reference, Key, Authors, Editors, Name, First_Name, Last_Name* $\}$, that can be defined for BibTeX files. The expression

$$(Reference \supset Authors \supset \sigma_{\text{``}Chang\text{''}}(Last\_Name)) \cup \\ (Reference \supset Editors \supset \sigma_{\text{``}Corliss\text{''}}(Last\_Name))$$

returns the set of $Reference$ regions that either contain an $Authors$ region containing a $Last\_Name$ region that is the word "Chang", or contain an $Editors$ region containing a $Last\_Name$ region that is the word "Corliss".

3.2 Optimizing Region Expressions

We begin by describing some of the properties of expressions in the region algebra that are the basis of the optimization technique, and exploited throughout the rest of the paper.

Our goal is to translate database queries to region expressions, and evaluate them using the indexing engine. Note that database queries may be expressed by several different region expressions, some of which are more efficient, and some less. Clearly, we want to evaluate the query using the most efficient expression. We therefore present below an optimization algorithm that given such an expression, finds the most efficient equivalent expression. In the rest of this section we concentrate on region expressions where all the operations are $\supset$ and $\supset_d$. (As we show later, this type of expressions are used for evaluating database queries on text files). We call such expressions **inclusion expressions**.

We first observe that files of a specific format have specific inclusion relationships among regions. For instance, in our BibTeX file example, $Reference$ regions can include $Editors$ region, but not vice versa. To describe such relationships between regions, we introduce a **region inclusion graph** (*RIG*, for short). The nodes of the graph are region names, and the edges state the possible inclusion relationships between the corresponding region instances. An edge (R_i, R_j) is in the graph, iff an R_i region can directly include an R_j region. The graph is used to characterize a set of instances that obey certain inclusion restrictions. In general, the RIG may contain cycles (e.g., self-nested regions).

Definition 3.1 *An instance I of a region index $\mathcal{I} = \{R_1, \ldots, R_n\}$* satisfies *a RIG (region inclusion graph) $G = (\mathcal{I}, E)$ iff for every two regions $r_i \in R_i(I), r_j \in R_j(I)$, if r_i directly includes r_j then $(R_i, R_j) \in E$. The set of all instances of $\mathcal{I}$ that satisfy a RIG G is denoted $\mathcal{I}_G$.*

We next consider equivalence of region expressions. In the standard database approach, two queries over a given schema are equivalent iff they have the same result for every instance of the database. In the context of queries in the region algebra, a RIG can be viewed as schema. We therefore have the following definition.

Definition 3.2 *Two region expressions e_1, e_2 are* **equivalent** *with respect to a RIG $G = (\mathcal{I}, E)$ iff for every instance $I \in \mathcal{I}_G$, $e_1(I) = e_2(I)$.*

For example, let $\mathcal{I} = \{$ *Reference, Key, Authors, Title, Editors, Name, First_Name, Last_Name* $\}$, and consider the following RIG.

Consider, the expression $e_3 = Reference \supset Title \supset Last\_Name$. The result of e_3 is empty for all the instances satisfying the above inclusion graph. This is because in all those instances, no *Last_Name* region is included in a *Title* region. In general we have that

Proposition 3.3 *Let $\mathcal{I} = \{R_1, \ldots, R_n\}$ be a region index, and let e be an inclusion expression over $\mathcal{I}$. Let $G = (\mathcal{I}, E)$ be a region inclusion graph.*
$e(I) = \emptyset$ for every $I \in \mathcal{I}_G$, iff at least one of the following holds:
(i) e has a subexpression $R_i \supset_d R_j$, and $(R_i, R_j) \notin E$.
(ii) e has a subexpression $R_i \supset R_j$, and G does not contain a path from R_i to R_j

The proof follows immediately from the properties of the instances $I \in \mathcal{I}_G$ that satisfy G.

We next show that knowledge about the structure of files can be used to shorten inclusion expressions, and to replace the $\supset_d$ operation by $\supset$. W.l.o.g we consider in the following only non trivial expressions (i.e. expressions whose result is not always empty).

Definition 3.4 *Let $G = (\mathcal{I}, E)$ be a RIG, and let e_1, e_2 be two inclusion expressions over $\mathcal{I}$. We say that e_2 is* **more efficient** *than e_1 w.r.t. G, iff e_1 and e_2 are equivalent w.r.t. G, and e_2 was obtained from e_1 by replacing sub-expressions of the form $R_{i_1} \circ_1 R_{i_2} \ldots \circ_{n-1} R_{i_n}$ (where $\circ_i$ is $\supset$ or $\supset_d$), by $R_{i_1} \supset R_{i_n}$. We say that e_2 is the* **most efficient** *version of e_1 w.r.t. G, iff it is more efficient than e_1, and there is no other expression that is more efficient than e_2 w.r.t. G.*

We show below that every inclusion expression e_1 has a unique most efficient version. Furthermore, we present an algorithm that computes this expression in time polynomial in the size of e_1. The algorithm is based on the following observations

Proposition 3.5 *Let $\mathcal{I} = \{R_1, \ldots, R_n\}$ be a region index. Let $G = (\mathcal{I}, E)$ be a RIG. Let e, e_1, e_2 be inclusion expressions over $\mathcal{I}$, s.t. e_1 is constructed from e by replacing the rightmost subexpression $R_i \supset_d R_j$ by $R_i \supset R_j$, and e_2 is constructed from e by replacing some subexpression $R_i \supset R_j \supset R_k$ by $R_i \supset R_k$.*
(a) e_1 is equivalent to e w.r.t. G, iff every path from R_i to R_j in G has the edge (R_i, R_j) and if there is a cycle through R_i then $\supset$ precedes R_i in e.
(b) e_2 is equivalent to e w.r.t. G, iff every path from R_i to R_k in G passes through R_j.

The proposition is proved by analyzing the cases that can cause ambiguity in the interpretation of inclusion relationship between regions (details are omitted for lack of space).

The Optimization Algorithm

We present below an optimization algorithm, that given an inclusion expression e computes the most efficient version of e. The algorithm has two steps. The first replaces $\supset_d$ operations by $\supset$, and the second shortens the expression.

1. The rightmost subexpression $R_i \supset_d R_j$, satisfying criteria (a) of Proposition 3.5, is replaced by $R_i \supset R_j$, and this step is repeated on the resulting expression until no more changes can be done.

2. All subexpressions of the form $R_i \supset R_j \supset R_k$, satisfying criteria (b) of Proposition 3.5, are replaced by $R_i \supset R_k$, and this step is repeated until no more changes can be done.

When the algorithm is applied on the expression $Reference \supset_d Authors \supset_d Name \supset_d Last\_Name$, the first step replaces the three $\supset_d$ operations by $\supset$ from right to left. The second step replaces $Authors \supset Name \supset Last\_Name$ by $Authors \supset Last\_Name$, obtaining $Reference \supset Authors \supset Last\_Name$. No more simplifications can be done due to the multiple paths from *Reference* to *Last_Name*. (The inclusion of *Last_Name* in *Authors* must be tested to filter out last-names of editors.)

Theorem 3.6 .
(i) Every inclusion expression e has a unique most efficient version e'.
(ii) The optimization algorithm, on input e, computes this e', in time polynomial in the size of e.

Proof: (sketch) We prove the theorem by showing that

- only the kind of rewriting done by the algorithm can yield a more efficient expression which is still equivalent to the original one,

- the replacement system used by the algorithm satisfies the finite Church-Rosser property (this is shown using Sethi's theorem [Set74]).

In the following sections we present a technique for translating database queries on files, into inclusion expressions. The expressions are then optimized using the above algorithm and evaluated using the indexing engine.

4 Mapping Files to Databases

In this section we consider mappings between files and databases, and explain how to use such mappings for deriving a RIG for regions in a file.

4.1 Structuring Schemas

Structuring schemas were introduced in [ACM93] as a tool for specifying how the data stored in a file should be interpreted in a database. Structuring schemas enable users to view information stored in files as if it is stored in a database, and to use database query and update languages for accessing this information. We briefly describe below the main concepts. For full discussion see [ACM93]. In the sequel, we assume standard knowledge on object-oriented databases, context-free grammars and parsing.

A *structuring schema* consists of a database schema and a grammar annotated with database programs. The grammar describes some of the structure of the file. The annotation specifies the relationship between the grammar non-terminals and their database representation. In particular, it specifies how a word w derivable from a nonterminal A should be represented in a database. This is done by associating to each derivation rule $A \rightarrow A_1, \ldots, A_n$ a statement describing how the database representation of a word derived from this rule is constructed using the database representations of the subwords derived from $A_1, \ldots, A_n$.

In the sequel, we use a Yacc-like notation [AJ74]. In a rule $A \rightarrow A_1, \ldots, A_n$, \$i denotes the database image of the string corresponding to A_i, and \$\$ the one associated to A.
The next example provides a *simplistic subset* of the structuring schema for BibTeX files. Every BibTeX file is represented in the database as a set of reference objects. Each such object has attributes containing the key of the reference, the title, the set of authors, etc.

The first part of the specification defines the classes and types used in the database representation. The second part associates with each non-terminal in the grammar the type/class used for representing words derived from that non-terminal. The third part describes how the words are mapped into their database representation.

`/* Classes and types */`

$$
\begin{array}{ll}
Class & Reference = \\
 & tuple(Key : string, Authors : set(Name), \\
 & Title : string, \ldots, Editors : set(Name), \ldots) \\
Type & Name = \\
 & tuple(First\_Name : string, Last\_Name : string)
\end{array}
$$

`/* Non-terminals type definition */`

$$
\begin{array}{llll}
Type & \langle Ref\_set \rangle & = & set(Reference) \\
Type & \langle Reference \rangle & = & Reference \\
Type & \langle Key \rangle & = & string \\
Type & \langle Authors \rangle & = & set(Name) \\
Type & \langle Title \rangle & = & string \\
Type & \langle Editors \rangle & = & set(Name) \\
Type & \langle Name \rangle & = & Name \\
Type & \langle First\_Name \rangle & = & string \\
Type & \langle Last\_Name \rangle & = & string
\end{array}
$$

`/* Annotated grammar` $BibTeX\_Schema$ `*/`[3]

$$
\begin{array}{lll}
\langle Ref\_Set \rangle & \rightarrow & \langle Reference \rangle^* \\
 & & \{\$\$:= \bigcup \$i\} \\
\langle Reference \rangle & \rightarrow & \text{“@INCOLLECTION\{”} \ \langle Key \rangle \\
 & & \text{“AUTHOR = ”} \ \langle Authors \rangle \\
 & & \text{“TITLE = ”} \ \langle Title \rangle \ldots \\
 & & \text{“EDITOR = ”} \ \langle Editors \rangle \ldots \} \\
 & & \{\$\$:= new(Reference, tuple(Key : \$1, \\
 & & \quad Authors : \$2, \\
 & & \quad Title : \$3, \ldots \\
 & & \quad Editors : \$6, \ldots))\} \\
\langle Key \rangle & \rightarrow & string \\
 & & \{\$\$:= \$1\} \\
\langle Authors \rangle & \rightarrow & \langle Name \rangle^* \\
 & & \{\$\$:= \bigcup \$i\} \\
\langle Title \rangle & \rightarrow & string \\
 & & \{\$\$:= \$1\} \\
\langle Editors \rangle & \rightarrow & \langle Name \rangle^* \\
 & & \{\$\$:= \bigcup \$i\} \\
\langle Name \rangle & \rightarrow & \langle First\_Name \rangle \ \langle Last\_Name \rangle \\
 & & \{\$\$:= tuple(First\_Name : \$1, \\
 & & \quad Last\_Name : \$2)\} \\
\langle First\_Name \rangle & \rightarrow & string \\
 & & \{\$\$:= \$1\} \\
\langle Last\_Name \rangle & \rightarrow & string \\
 & & \{\$\$:= \$1\}
\end{array}
$$

Structuring schemas can be used to specify a virtual database view over files [ACM93]. To answer a query on the database view of a file, one may construct the database image of the file (i.e. parse the file using the structuring schema, construct the objects/tuples, and load them into the database), and then evaluate the query on the database. This technique will obviously lead to scanning and parsing the whole file, and constructing many unnecessary objects and complex values. This is time and space consuming and we want to avoid it. The optimization technique presented in [ACM93] reduces the amount of data loaded into the database while answering a query. But the whole file still needs to be scanned and parsed. We will show below that this unnecessary effort can be

[3] We use the notation $A \rightarrow B^* \ \{\$\$:= \bigcup \$i\}$, to denote the fact that A is a sequence (possibly empty) of B's, and that the database representation of A is a set containing the database representation of all the B's in the sequence.

avoided using text indexing techniques. The key observation is that word and region indices can be used to locate substrings that are potentially relevant to the query computation, and thus save on scanning the whole file when evaluating the query. Moreover, we identify cases where database queries on a file can be fully computed using the indexing engine, and the scanning of the file can be completely avoided.

4.2 Deriving a RIG from a Natural Structuring Schema

Note that the database representation of the BibTeX file is rather "natural", i.e. it is very close to the actual structure of a file. We call such structuring schemas **natural schemas**. The database representation using a natural schema is essentially the *p-string* of [GT87]. In general, the database representation may significantly differ from the file structure (for examples, see [ACM93]).

To simplify the presentation we demonstrate the optimization technique on queries over views defined using natural structuring schemas. We assume below that literals A defined using rules of the form $A \to B^*$ are represented in the database by sets or lists. Literals A defined using rules of the form $A \to B_1 \dots B_n$ are represented by tuples or by objects whose attributes correspond to $B_1 \dots B_n$. We also assume that the names of attributes are the same names of the non-terminals they represent[4]. Terminals are represented by atomic types[5].

We first consider a simple case where all the words in the file are being indexed, and where the region index $\mathcal{I} = \{A_1, \dots, A_n\}$ contains all the non-terminal names A_i in the grammar G, except the root of the grammar. We defer the discussion of partial indexing to section 6). We also assume that each index A_i is instantiated by the set of all regions corresponding to occurrences of A_i in the parse tree of the file f using the grammar G.

The inclusion relationship between the indexed regions is determined by the grammar. In particular, a region a_i corresponding to a nonterminal A_i can directly include a region a_j corresponding to a non-terminal A_j iff the grammar G has a rule where A_i appears of the left side, and A_j on the right side.

[4]This requires that every non-terminal name appears at most once in the right hand side of a rule. This is not a serious limitation since every grammar can be easily adjusted to satisfy this requirement.

[5]When considering general context-free grammar, *disjunctive types* will naturally arise from non terminals defined disjunctively. This may not be realized directly is some database systems. There are of course a variety of means of simulating such types (in particular using inheritance).

Thus, the region inclusion graph of $\mathcal{I}$ can be automatically derived from the grammar G. The nodes are the non-terminals of the grammar, and the graph has an edge (A_i, A_j) iff G has a rule where A_i appears as the left side, and A_j as the right side. For example, the graph in Section 3.2 is part of the RIG defined by the BibTeX grammar.

It is important to note that the optimization technique presented in the following sections is applicable to other mappings between files and databases as well. The technique depends only on the existence of a region inclusion graph describing the relationships between the indexed regions, and on the existence of a mapping between path expressions in queries and paths in the region inclusion graph. In the case of natural structuring schemas, the graph and the mapping can be automatically derived from the grammar. In case of more general mappings, the user may need to provide this information as part of the database schema.

5 Querying Fully Indexed Files

This section describes how to translate database queries into expressions in the region algebra, assuming that all the needed regions are indexed. The last subsection highlights a class of queries that are very expensive when computed in traditional databases, but are significantly cheaper using text indexing.

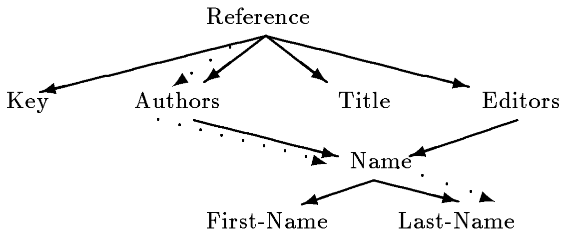
5.1 From Simple Queries to Region Expressions

We first study optimization of simple queries of the form "`SELECT` r `FROM` R `WHERE` $r.p = w$", where p is some path expression, and R is a database view of a file f defined using a natural structuring schema with a grammar G.

Let *References* be a view defined using the BibTeX structuring schema. Consider the query

$$Q = \texttt{SELECT } r \texttt{ FROM } References\ r \quad \texttt{WHERE } r.Authors.Name.Last\_Name = \text{``}Chang\text{''}$$

The references retrieved by this query, are exactly those that match *Reference* regions that directly include an *Authors* region, that directly include a *Name* region, that directly include a *Last_Name* region, that contains exactly the string "Chang". In fact, the path expression in the query corresponds to a path in the RIG of the fully indexed file. Part of the RIG is presented below. The path is denoted by dashed arrows.



Consider the parse tree of the BibTeX file, shown in Figure 5.1. The data used to answer the query resides in regions reachable by paths that match the path in the RIG.

It follows that the references retrieved by the query Q can be selected using the expression $e1 = Reference \supset_d Authors \supset_d Name \supset_d \sigma_{\text{“}Chang\text{”}}(Last\_Name)$. As shown in section 3.2, this expression can be optimized, obtaining $e2 = Reference \supset Authors \supset \sigma_{\text{“}Chang\text{”}}(Last\_Name)$. To compute the query, we: (i) evaluate e_2, (ii) parse the reference regions in the result using the BibTeX structuring schema, obtaining reference objects, and (iii) return these objects to the user.

Recall that we consider here only natural structuring schemas. This implies that every path expression p in the query, matches a derivation sequence(s) in the grammar (p may match several derivation paths due to conjunctive rules). This also implies that the attributes used in the path p match regions that correspond to the nonterminals in this derivation. In general, every path p in a query "SELECT r FROM R WHERE $r.p = w$", matches path(s) $A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_m$ in the RIG. The path(s) can be easily determined by syntactically analyzing the grammar. The regions matching the objects retrieved by Q are exactly those regions selected by the inclusion expressions $A_1 \supset_d A_2 \supset_d \ldots \supset_d \sigma_w(A_m)$

Thus to compute the query efficiently we (i) transform the query to an inclusion expression, (ii) optimize the inclusion expression, (iii) evaluate the inclusion expression, (iv) parse the resulting regions, and return the required objects.

5.2 Select–Project–Join Queries

The queries considered above have only one selection criterion comparing an attribute to a constant. We next consider queries with more complex selection criteria. Consider first selections that compare the values of two attributes. The query

$Q =$ SELECT r FROM $References\ r$
WHERE $r.Editors.Name. = r.Authors.Name$

selects references that appeared in books edited by one of the authors. Unlike the queries discussed in the previous subsections, this query cannot be fully evaluated using the region algebra. The problem is that the region algebra does not support operations comparing contents of regions. This limitation is typical to text indexing systems. It signals out the difference between database systems and traditional text indexing systems, and the benefits one gains from having a full database interface to files.

It turns out, however, that indices can still be used to accelerate the computation. The region index can be used to locate the regions corresponding to the attributes specified by the two paths. The content of the regions is then loaded into the database, and a database join operator is used to select regions with matching content. Then, the region index is used again to locate the references containing those matching strings.

Queries may select elements based on several selection criteria composed using *and*, *or* and *not* operators. These operations can be simulated in the region algebra using *union*, *intersection* and *subtraction* of the corresponding index expressions. Note that different selection criteria may access common attributes. As in classical query optimization, the goal is to find common subexpressions in the region expressions and evaluate them once.

Projection is handled similarly to selection. Instead of using the $\supset$ and $\supset_d$ operators, we use the $\subset$ and $\subset_d$, resp. For example, the query

$Q =$ SELECT $r.Authors.Name.Last\_Name$
FROM $References\ r$

is translated to $e_1 = Last\_Name \subset_d Name \subset_d Authors \subset_d Reference$.

The optimization technique presented in section 3.2 works for expression containing $\subset$ and $\subset_d$ operations as well. In particular, e_1 above is optimized getting $e_2 = Last\_Name \subset Authors \subset Reference$.

Complex queries involving several view definitions (e.g. the BibTeX authors that are cited in a LaTex file) or several occurrences of the same view (e.g. nested queries), use join. Text indexing systems are inadequate for performing join like computation. This must be done at the database level. However, we can use the indexing system to reduce the amount of information loaded to the databases for performing the join. The idea is to use rewriting rules to push selection and projection down as much as possible, and then use indexes to locate the elements needed for the join computation. We will not address in details the problems raised because they are similar to that of complex queries in any rewriting system.

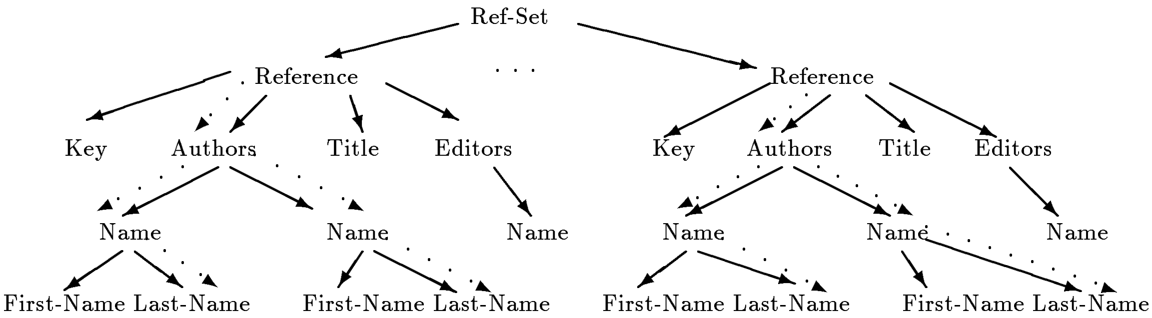


Figure 2: The parse tree for BIBTEX files (full indexing).

5.3 Extended Path Expressions

Information stored in files often has complex structure because it represents objects that are inherently complex. It has been observed [KKS92, MBW80] that simple path expressions are not always suitable for manipulating objects with complex structure. One way to overcome this problem is to use path expressions with variables.

Assume that one wants to find all references where "Chang" is an author or an editor. This can be expressed by the XSQL query

$$Q = \texttt{SELECT}\ r\ \texttt{FROM}\ References\ r$$
$$\texttt{WHERE}\ r.*X.Last\_Name = \text{“Chang”}$$

The notation $*X$ means that we are interested in the attribute *Last_Name*, no matter what is the path leading to this attribute. A similar facility for querying text databases with partial knowledge of the schema is described in [KM93].

A naive way to evaluate such a query is to analyze the path expression, find all the possible assignments to the variables, and then evaluate the query separately for each specific path. There are cases, however, where a better evaluation strategy exists. In particular, if a variable name $*X$ appears in the path expression only once, then any path from the attribute left of $*X$ to the attribute on the right is an acceptable assignment. Recall that the database representation of files naturally matches their structure, and that attributes correspond to regions in the file. The attribute A_j specified by the path expression $A_i.*X.A_j$ resides in a region of type A_j that is included in some region of type A_i. Thus, $A_i.*X.A_j$ (where X appears in p only once) is translated to $A_i \supset A_j$. Variables that appear more than once need to be instantiated by appropriate attribute sequences, and are translated as before (using the $\supset_d$ operator). Thus, the query Q above can be mapped into the region algebra expression $Reference \supset \sigma_{\text{“}Chang\text{”}}(Last\_Name)$, which can be evaluated much more efficiently than in the standard approach.

XSQL also supports path expressions of the form $A_i.X_1.X_2 \ldots X_n.A_j$ (i.e variables without the star notation). This is used to access A_j attributes that are reachable from A_i by an arbitrary path of length n. (In contrast, the star notation denotes paths of arbitrary length). This can be simulated in the region algebra by looking for regions a_1 of type A_1 that contain some region a_2 of type A_2, provided that there are exactly i nested regions contained in a_1 and containing a_2.

It is important to emphasize that in traditional OODBMS, path expressions with variables are computationally more expensive than those with no variables (since the system has to actually traverse all possible paths). In contrast, for text files, path expressions with variables may be cheaper. This is due to the fact that simple inclusion ($\supset$) may be applicable instead of direct inclusion ($\supset_d$).

One could also go beyond first order queries, and use a notation borrowed from GraphLog [Con89, CM90]: *path regular expressions.* These extend path expressions with the traditional regular expression operators (in particular, the transitive closure operator). Within the framework we describe here it is possible to evaluate paths with a regular expression involving a transitive closure, with just an inclusion expression. This shows, once more, that in some cases a traditionally expensive query (a closure) can be implemented much more efficiently with the techniques we describe here.

6 Partial Indexing

In the previous sections we assumed that all the nonterminals in the grammar are indexed. In practice, we may want to create a smaller number of region indexes to reduce the space and update costs. We show below that performance improvements can be obtained even when only a selected subset of regions is indexed.

Partial indexing may not be sufficient for fully evaluating queries using the region algebra. It can be used, however, to significantly reduce the search space. In the presence of partial indexing, a query is computed in two phases: (i) The query is compiled into an inclusion expression that computes a super set of the required result - a set of *candidate regions*, and (ii) the candidate regions are further processed to obtain the exact result.

6.1 Obtaining the Candidate Regions

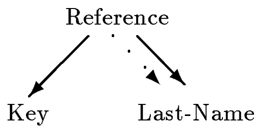
In this subsection we describe the first phase – transforming the query to an inclusion expression computing a set of candidate regions. The second phase - the regions processing – is discussed next.

Let $\mathcal{I}_p = \{A_{i_1}, \ldots, A_{i_k}\}$, be a region index containing part of the non-terminal names in the grammar G. Assume that each index A_{i_j} is instantiated by the set of all regions corresponding to occurrences of A_{i_j} in the parse tree of the file f using the grammar G.

As in the case of full indexing, the inclusion relationship between the indexed regions is determined by the grammar. In particular, the region inclusion graph of $\mathcal{I}_p$ can be automatically derived from the grammar G. The nodes are the indexed non-terminal in $\mathcal{I}_p$. The graph has an edge (A_i, A_j) iff in the RIG of the full grammar (i.e. where all the non-terminals are indexed) there is a path from A_i to A_j where all the non-terminals on the path other than A_i,A_j are not indexed (i.e. do not belong to $\mathcal{I}_p$).

For example consider the BibTeX grammar, and let $\mathcal{I}_p = \{Reference, Key, Last\_Name\}$. The corresponding RIG is presented below. As before, a path p in a database query matches a path in the RIG. The dashed arrows in the following diagram correspond to the path in the query

$$Q = \texttt{SELECT}\ r\ \texttt{FROM}\ References\ r$$
$$\texttt{WHERE}\ r.Authors.Name.Last\_Name = \text{“}Chang\text{”}$$



Once more, consider the parse tree of the BibTeX file. The data used to answer the query, resides in regions reachable by paths that match the path in the RIG. In the case of full indexing, the indexed information was sufficient for exactly locating the regions needed for the query processing. In the case of partial indexing only an approximation of the required regions can be achieved. Below is a part of the parse tree of a bibtex file. The dashed lines in Figure 6.1 correspond to paths that match the path in the above RIG.

Note that due to the partial indexing, one cannot distinguish between last names of authors and last-names of editors. Thus, the inclusion expression $Reference \supset_d \sigma_{\text{“}Chang\text{”}}(Last\_Name)$ identifies a superset of the required references, (references where “Chang” is either an author or an editor).

In general, a path p in a query

$$Q = \texttt{SELECT}\ r\ \texttt{FROM}\ References\ r\ \texttt{WHERE}\ r.p = w$$

matches a path $A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_m$ in the RIG of the indexed non-terminals. The inclusion expression $A_1 \supset_d A_2 \supset_d \ldots \supset_d \sigma_w(A_m)$ retrieves a set of candidate regions, that is a superset of the regions required by the query. (This expression can be further optimized using the optimization algorithm of Section 3.2).

Note that there are cases where the set of candidate regions coincide with the query's answer. The conditions under which this happens are discussed in Section 6.3.

6.2 Parsing the Candidate Regions

We next filter out irrelevant the regions. To this end, we parse the regions in the superset, building for each region a corresponding database representation, and then select the required elements by applying the query on the resulting database objects.

It was observed in [ACM93] that the structuring schema can be optimized by “pushing” the query into the parsing process, so that only objects that meet the query selection criteria are built. Parsing using an optimized schema reduces the construction of unnecessary database objects.

6.3 Exact Answer with Partial Indexing

There are cases where partial indexing is sufficient for fully computing the query, without additional parsing. This happens when the indexed non-terminals provide enough information to avoid ambiguities in path interpretation. The conditions are sketched below.

Let $\mathcal{I}$ be a region index containing all the names of non-terminals in the grammar G, and let $\mathcal{I}_p \subseteq \mathcal{I}$ be a partial index. Let $RIG(\mathcal{I})$ and $RIG(\mathcal{I}_p)$ be the corresponding region inclusion graphs. The key observation is that every edge (A_i, A_j) in $RIG(\mathcal{I}_p)$ matches path(s) from A_i to A_j in $RIG(\mathcal{I})$, where all

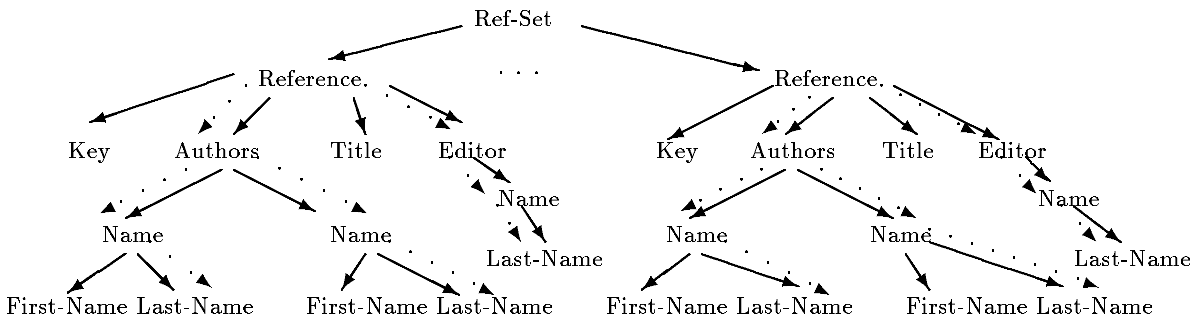


Figure 3: The parse tree for BibTeX files (partial indexing).

the nodes on the path(s), other than A_i and A_j, are not in $\mathcal{I}_p$.

Consider a query "`SELECT` r `FROM` R r `WHERE` $r.p = w$". Let $A_1 \to A_2 \to \ldots \to A_n$ be the path in $RIG(\mathcal{I}_p)$ corresponding to the path p in the query Q. The inclusion expression $A_1 \supset_d A_2 \supset_d \ldots \supset_d \sigma_w(A_n)$ fully computes Q iff each of the edges (A_i, A_j) on this path matches a unique path in $RIG(\mathcal{I})$. If the edges match several paths, then the inclusion expression computes a superset of the required regions.

7 Choosing What to Index

The efficiency of query evaluation depends on the choice of region indices. There is a tradeoff between performance and the number of regions being indexed. As in standard database systems, knowledge about the access patterns to files can be used to select indices. In fact, many aspects of this issue should be considered in the broader context of indexing techniques for OODBMS's (see [Ber94] for a survey). A key observation in our context is that in many cases partial indexing is sufficient for full computation of queries.

Consider a query "`SELECT` r `FROM` R r `WHERE` $r.p = w$", where R is a view defined by a structuring schema with grammar G. Assume first that all the non-terminals in the grammar are indexed, and let $e = A_1 \circ_1 A_2 \circ_2 \ldots \circ_{n-1} A_n$ be the optimized inclusion expression that computes Q (e is constructed and optimized as explained in Section 5).

Note that not all the indexed non-terminals are actually needed for evaluating e. One can distinguish two kinds of region indices used in the computation. Regions that are explicitly mentioned in e (i.e. the A_i's), and those that are implicitly mentioned. The implicit usage is due to the $\supset_d$ operation. To compute $A_i \supset_d A_{i+1}$, one has to rule out all non direct inclusions. This requires checking that none of the other indexed regions resides between the two regions. Note however that not all the indexed regions must indeed be checked. The grammar G enforces certain relationships between regions. In fact, only regions corresponding to non-terminals derivable from A_i and deriving A_j can violate the direct inclusion. Thus only region indices corresponding to those non-terminals need to be checked. Moreover, if A_i derives some $A_{i'}$ that derives some $A_{i''}$ that derives A_{i+1}, it is not necessary to check both $A_{i'}$ and $A_{i''}$, one suffices.

In summary, to fully compute Q, it is sufficient to (i) index the nonterminals mentioned in e, and (ii) for every subexpression $A_i \supset_d A_{i+1}$ in e, index one non-terminal (other than A_i,A_{i+1}) on each path from A_i to A_{i+1} in the RIG of the grammar G.

Indexing can be either performed globally (i.e., for the whole file) or only in specific regions. For example, assume that users often query names of authors, but never (or hardly ever) query names of editors. In that case, instead of indexing all the *Name* regions it is better to index only those that reside in some *Authors* region.

As explained in Section 6, one can trade indexing for accuracy of computation. The parameters taken into consideration are the number of regions needed to be indexed for full computation, and the number and expected size of the regions needed to be parsed due to non indexed data.

8 Conclusions

In this paper we discussed how to provide efficient access to semi-structured data residing in files. The approach combines the convenience of using an extended SQL to query the more structured components of the information in files with the efficiency afforded by text indexing.

An original contribution of the paper consists of an optimization technique that is applicable when advanced text indexing technology is available to the database query evaluator. The concept of a region inclusion graph is introduced to provide the information needed to perform the above optimization. We also discuss how to automatically derive a RIG when structuring schemas are used to specify the mapping from files to databases.

Preliminary experimental results show that significant performance improvements can be obtained by using optimized PAT inclusion expressions (instead of relying on a traditional database engine) for the evaluation of queries.

Acknowledgments: We would like to thank Vassos Hadzilacos for his fruitful suggestions, and Frank Tompa and Pekka Kilpeläinen for detailed comments on a preliminary version of this paper. The first author would like to acknowledge Gaston Gonnet for several discussions on PAT. This work was done at University of Toronto and the second author was supported by the Institute for Robotics and Intelligent Systems.

References

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. of the 19th Int. Conf. on Very Large Databases, VLDB93*, pages 73–84, 1993.

[AJ74] A. V. Aho and S. C. Johnson. Programming utilities and libraries LR parsing. *Computing Surveys*, June 1974.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. Query languages for object-oriented database systems: the O2 proposal. In *Proc. DBPL, Salishan Lodge, Oregon*, June 1989.

[Ber94] E. Bertino. A Survey of Indexing Techniques for Object-Oriented Database Management Systems. In Freytag, J. and Maier, D. and Vossen, G., editor, *Query Processing for Advanced Database Systems*, pages 383–418, San Mateo, CA, 1994. Morgan Kaufmann.

[BGH+92] T. F. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W. H. Mansfield, J. Raitz, and A. Weinrib. The Datacycle architecture. *Communications of the ACM*, 35(12):71–81, December 1992.

[BGMM93] D. Barbara, H. Garcia-Molia, and S. Mehrota. The gold mailer. In *IEEE Data Eng.*, pages 92–99, 1993.

[Bur92] F. J. Burkowski. Retrieval activities in a database consisting of heterogeneous collections of structured text. In *Proc. of the 15th. SIGIR Conference*, pages 112–125, 1992.

[CM90] M. Consens and A. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[CM93] M. Consens and A. Mendelzon. Hy: A hygraph-based query and visualization system. In *Proceedings of the ACM-SIGMOD 1993 Annual Conference on Management of Data*, pages 511–516, 1993.

[Con89] Mariano P. Consens. Graphlog: "real life" recursive queries using graphs. Master's thesis, Department of Computer Science, University of Toronto, January 1989.

[GNOT92] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry. Using collaborative filtering to weave an information tapestry. *CACM*, 35(12), December 1992.

[Gon87] G. Gonnet. Examples of PAT applied to the Oxford English Dictionary. Technical Report OED-87-02, University of Waterloo, 1987.

[GT87] G. Gonnet and F. Tompa. Mind your grammar: a new approach to modelling text. In *Proc. of the 13th Int. Conf. on Very Large Databases*, pages 339–346, 1987.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. SIGMOD, San-Diego*, 1992.

[KM93] P. Kilpeläinen and H. Mannila. Retrieval from hierarchical texts by partial patterns. In *Proc. of the 15th. SIGIR Conference*, 1993.

[Lam85] L. Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley, Reading, MA, 1985.

[MBW80] J. Mylopoulos, P. A. Bernstein, and H. K. T Wong. A language facility for designing database-intensive applications. *ACM Transactions on Database Systems*, 5(2), 1980.

[Ope93] Open Text Corporation. *PAT Reference Manual and Tutorial*, 1993.

[Pae93] A. Paepcke. An object oriented view onto public hetrogeneous text databases. In *IEEE Data Eng.*, page 484, 1993.

[Sch93] M. F. Schwartz. Internet Resource Discovery at the University of Colorado. *IEEE Computer Networking*, 26(9), September 1993.

[Set74] R. Sethi. Testing for Church-Rosser Property. *JACM*, 21(4), October 1974.

[SLS+93] K. Shoens, A. Luniewski, P. Schwartz, J. Stamos, and J. Thomas. The Rofus system: Information organization for semi-structured data. In *Proc. of the 19th Int. conf. on Very Large Databases, VLDB 93*, pages 97–107, 1993.

[SM83] G Salton and M. J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, 1983.

[ST92] A. Salminen and F. W. Tompa. PAT expressions: an algebra for text search. In *Papers in Computational Lexicography: COMPLEX'92*, pages 309–332, 1992.

[Yeu93] A. Yeung. Text Searching in the Hy^{+} Visualization System. Master's thesis, Department of Computer Science, University of Toronto, October 1993.