Join Queries with External Text Sources: Execution and Optimization Techniques

Surajit Chaudhuri, Umeshwar Dayal, Tak W. Yan*
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
Email: lastname@hpl.hp.com

Abstract

Text is a pervasive information type, and many applications require querying over text sources in addition to structured data. This paper studies the problem of query processing in a system that loosely integrates an extensible database system and a text retrieval system. We focus on a class of conjunctive queries that include joins between text and structured data, in addition to selections over these two types of data. We adapt techniques from distributed query processing and introduce a novel class of join methods based on *probing* that is especially useful for joins with text systems, and we present a cost model for the various alternative query processing methods. Experimental results confirm the utility of these methods. The space of query plans is extended due to the additional techniques, and we describe an optimization algorithm for searching this extended space. The techniques we describe in this paper are applicable to other types of external data managers loosely integrated with a database system.

1 Introduction

A growing number of text information sources are available today. These range from traditional library information systems such as Library of Congress LOCIS [Lib94] and Stanford University FOLIO, to online services such as Dialog, and eventually to comprehensive digital libraries [Cor94, CMU94]. Many applications and end users need to combine the retrieval of text information from these sources with structured data retrieved from a database system. For example, [YA94] describes a hospital information system that permits physicians to access progress notes, medical literature, and drug formularies, in addition to structured data from the patient's medical record stored in one or more database systems.

*Also: Department of Computer Science, Stanford University, Stanford, CA 94305

Database systems provide powerful query languages for structured data, but are not particularly well suited for storing or querying text information. SQL provides only primitive string matching operations. Text retrieval systems, on the other hand, use indexing techniques and processing algorithms that are specialized for querying text information, but are not suitable for storing and querying structured data. What is needed is a system that integrates these capabilities to support uniform ad hoc queries over structured data *and* text.

We can distinguish between "tight" and "loose" integration. Tight integration of a data type results in it being stored and processed within the database system, i.e., the access methods and evaluation methods are visible to the query processor (and may in fact be modified to match the query processing strategies). Loose integration of a data type assumes that it is managed by external data managers. Once these specialized data managers have been *registered* with the database system, queries can include operations (sometimes called "foreign functions") over this new (external) data type and operations supported by the database system. Loose integration is the only option available for accessing existing external data sources.

This paper studies the important problem of query processing in loosely integrated systems. There is previous work on general frameworks for extensible query processing and optimization (See Section 1.1). However, the problem of identifying query processing methods that are appropriate for operations that span different data types and repositories, and the impact of these methods on query optimization, have so far received little attention. For example, how should a join ("foreign join") between a relation and a collection of documents be processed? An obvious approach (and the one typically assumed) is to do *tuple substitution.* This corresponds to a nested loop join, with the relation as the outer operand and the document set as the inner. However, tuple substitution might be prohibitively expensive, since it involves repeated invocations of the external data manager. Do other query processing techniques from distributed database systems apply [BGWR81, LMH+85, YC85]? Are there new techniques that have not been described for distributed database systems?

The focus of this paper is on loose integration of a database system and a Boolean text retrieval system, which is the most common class of text system commercially available today. In section 2, we introduce the model of a Boolean text retrieval system, and define the query processing problem we have studied. Based on the characteristics of text systems, we develop (in Section 3) processing algorithms for joins involving a text system and a database system. In particular, we show that some distributed query processing methods such as semi-joins are applicable to this problem. We also develop a novel class of method based on "probing" that is especially applicable when the cost of invoking the text system is high. In Section 4, we develop a cost model for the various join methods. New issues arise when we consider more complex queries involving multiple joins between the text system and the database system. We define an extended execution space that provides additional opportunities for optimizing such queries. In Section 5, we describe an enumeration algorithm for efficiently exploring this extended execution space. In Section 6, we present experimental results that validate our approach with a real text retrieval system, CMU Mercury [CMU94]. Finally, in section 7, we discuss the lessons learned from this research. We discuss how the techniques generalize to other external data managers. We also identify features that might be added to text retrieval systems to make them better suited for integration with database query processing.

1.1 Related Work

There is recent work in several related areas. The problem of optimizing queries with complex selection conditions over structured documents has been addressed by [ACM93, CACS94, CM94]. Work on the tight integration with text retrieval has focused on the addition of new storage and access methods for text data within a database system [BRG88, LW90, LS88], and on optimizing selection queries on text data. Our work, in contrast, is on the loose integration with *external* text sources, and we consider queries that *join* relational and text data.

There also is work on the tight integration of spatial processing with a database system [AS91], which assumed that links (e.g., tuple identifiers) were maintained between the spatial and non-spatial components of an object; join methods using these links were described. Since we consider loose integration, we do not assume that such links are maintained.

Work on extensible query processing and optimization [BG92, CS93, HFLP89, GD87, KMP93, MDZ93, SJGP90] presented architectural frameworks and language support to allow the incorporation of new access and join methods (such as those proposed in this paper) in query processing and optimization. An optimization algorithm for queries with expensive selection predicates (but no foreign join) appears in [HS93]. The specific problem of query processing and optimization in the presence of foreign selections and joins was investigated in LDL [CGK89] and Papyrus [CHK+91, HHK+93, CS93]. However, the only join methods with foreign data types studied in the past were tuple substitution and its variants. We describe several alternative methods, tradeoffs in the choices and their impact on optimization algorithm. Our optimization algorithm is a simple extension of System-R style optimizer, making it possible to be adapted by existing systems that may not have extensible optimizers.

2 Background

2.1 Model of Text Retrieval Systems

There are three main classes of retrieval models found in current text retrieval systems: Boolean, vector space, and probabilistic [Sal89]. Of these, the Boolean model is the most common among commercial systems, and is the one we consider in this paper.

A text retrieval system manages a collection of documents, each of which is uniquely identified by a *docid*. A document consists of a set of text fields. In bibliographical systems, examples of text fields include author, title, abstract, date etc. The user locates documents of interest by issuing a `search`. A basic search term can be a word (e.g., `'filtering'`), a truncated word (e.g., `'filter?'`), or a phrase (e.g., `'information filtering'`). When there are more than one text fields, the search may be limited to a certain text field; e.g., search on author name:`AU='smith'`. Some systems support proximity searches (e.g., `'information' near10 'filtering'`.) These basic search terms can be combined to form complex search expressions using Boolean connectors `and`, `or`, and `not`; e.g., `'information filtering' and AU='smith'`.

Documents that exactly match a search expression are returned as the *result set*. This set contains the docids of matching documents and some of the text fields. Typically not all the text fields are returned, and the user may subsequently `retrieve` the entire document using its docid. We call the abbreviated format returned in a result set the *short form*, and the one retrievable by docid the *long form*.

To support fast searching, most text retrieval systems use access methods such as inverted indexes and signature files (see e.g., [Fal85, Sal89]). Inverted indexes are more appropriate in large-scale systems [Fal92]. Thus, we concentrate on inversion-based systems in this paper. In an inverted index, each word is associated with an inverted list of *postings* that record the docids of documents in which the word appears. A posting may also contain information such as the field the word occurs in and the position. We assume that the inverted lists reside on disk, and a main memory directory maps a word to the location of its list [DH91]. To process a search, inverted lists are retrieved and set operations are performed on the lists. For example, the search `'information' and 'filtering'` is performed by retrieving the lists for `'information'` and `'filtering'` and intersecting the lists. Typically the lists are sorted and set operations take time linear in the lengths of the lists [DH91, Sal89].

2.2 Querying Structured and Text Data

As discussed in Section 1, the foreign function capability supported in extensible database systems enables us to encapsulate external data servers such as text systems. We have implemented such an integration, tying an extensible database system, OpenODB [Hew92], with the Project Mercury server from Carnegie-Mellon University. Project Mercury [CMU94] is one of the ongoing digital library projects; its server makes available a number of text databases. We focused on the Computer Science Technical Report (CSTR) database, which contains bibliographic records from a consortium of universities. The underlying text retrieval engine uses the Boolean model described above.

To the user of the integrated system, an external text source appears as a relation. For ease of presentation, we describe our examples in SQL-like syntax. A document is a tuple and the fields are its attributes. For example, the relation `mercury` is defined as:

```
create table mercury
(docid varchar, title {varchar},
 author {varchar}, abstract {varchar}, ...)
```

The syntax for a text search predicate is <search term> in <field>; the predicate is true if the search term is found in the field. In general, the search term may be a single word or a phrase. We focus on Select-Project-Join queries (i.e., conjunctive queries) such as:

```
Q1: select * student, mercury
    where student.area = 'AI' and
          student.year > 3 and
          'belief update' in mercury.title and
          student.name in mercury.author
```

The query is essentially a "join" between the relational and document data on the names of senior AI students and the authors of Mercury documents that have the phrase 'belief update' in the title.

2.3 Problem Statement

We assume loose integration with an external Boolean text retrieval system, i.e., that the text retrieval system's internal data structures are not accessible to the database system, and there are no special structures in the text retrieval system that correlate with the relational database system. The database system accesses the text retrieval system via the `search` and `retrieve` operations described in Section 2.1. These assumptions imply that joins involving text data have to be executed as instantiated selections on the text retrieval system. We do assume that certain statistical meta information is extracted from the retrieval system (Section 4.2).

The problem can now be stated as:

> Given a conjunctive query that involves joins of structured data and external text data, construct a query plan to efficiently evaluate the query by sending to the text retrieval system one or more `search/retrieve` operations.

3 Alternatives for Single Join

In this section, we describe alternative ways of evaluating the simplest queries that have a single join between one relation and the text system. Henceforth, we refer to such a join as a *foreign join*.

3.1 Tuple Substitution

Traditionally, the *tuple substitution* (TS) technique is used to implement foreign joins with text systems (e.g., in [YA94]). Tuple substitution may be best viewed as a nested loop join where the relational table is the outer relation. Thus, every tuple of the relational table is instantiated in turn. For each instantiation, the join predicates on the text retrieval system turn into selection conditions for the text system. Such an instantiated query is now evaluable by the text system, as illustrated below:

Example 3.1: Consider Q1 in Section 2.2. For every instance of the `student` tuple, a query is invoked. Thus, for a `student` tuple that binds the value of `student.name` to `Radhika`, the corresponding query to Mercury obtained by the above tuple substitution is: `TI='belief update' and AU='Radhika'`.

Observe that the number of queries invoked is equal to the cardinality of the relational table. Therefore, for *large* relational tables, the tuple substitution method will be prohibitively expensive. The cost can be reduced somewhat by observing that for two queries obtained by tuple substitution to be different from each other, they must disagree in at least one join column. Therefore, we need only send a query for each *distinct* tuple in the projection of the relational table over the join columns. This can be achieved by either *caching* the values of join columns for previous queries, by exploiting an *existing order* on join columns or by grouping on the join columns [CS93].

In the next section, we will propose alternatives to tuple substitution that can take advantage of the structure of the query in additional ways. The first set of alternatives shows how techniques from distributed query processing can be fruitfully applied to our problem. Next, we introduce join methods based on *probing*, a new technique for doing join with text systems.

3.2 Relational Text Processing and Semi-join

In distributed query processing systems, it was observed that while joining two relations, there is a *choice of site* where the join could be done. SQL provides some, though limited, ability to do string processing. Therefore, if it happens that all the text join predicates require no more than SQL-supported string processing, then the relational engine provides an alternative to the text system to evaluate the predicates. Specifically, when the text predicates in the query are on short structured fields (e.g., author or title field), this alternative is attractive.

This method further requires that there are selection conditions on the text data. Relational Text Processing (RTP) proceeds as follows. First, a query that contains only the text selection conditions is sent to the text system. Next, the returned documents are matched with the relational tuples by a traditional join method (e.g., nested loop) using the string matching functions in SQL.

In contrast to tuple substitution where the number of queries sent to the text system is equal to the number of tuples in the relation (or, at least the number of distinct values in the join columns), this technique requires a *single* query to be sent to the text retrieval system. Thus, if the selection conditions on the text are *highly selective* and if the cost of sending a query to the text system is high, then relational processing could outperform tuple substitution.

Example 3.2: Applying this method to `Q1` in Example 3.1 results the following single query to Mercury: `TI='belief update'`. There were only a few bibliographic entries in Mercury that had `'belief update'` in the title. Since the text processing needed at the relational end was simple, relational processing does significantly better than tuple substitution (See Section 6). ∎

Semi-join:

In distributed databases, *semi-join* has been proposed as a technique to reduce the data transmission cost [BGWR81]. Semi-joins can be used to reduce the cost of a foreign join. First, assume that we are only interested in the set of matching docids and not in the relational attributes (i.e., the query itself is a "semi-join").

In TS, each relational tuple is turned into a conjunctive text search. The idea of the semi-join method (SJ) is to use the `or` connector and package a number of such conjuncts into a single search. Text retrieval systems typically allow a fairly large number of terms per search (e.g., Mercury allows 70) and thus this can significantly reduce the cost of foreign (semi-)joins.

Example 3.3: Consider the following semi-join query where we select the `docids` of reports only.

```
Q2: select docid from student, mercury
    where student.advisor = 'Garcia'
          and 'text' in mercury.title
          and student.name in mercury.author
```

Assume that the predicate `'text' in mercury.title` is not very selective. In that case, relational text processing is not an attractive alternative to tuple substitution. Suppose the set of Garcia's students is: `{Gravano, .. , Kao}`. In that case we can send a *single query* (assuming that there are fewer than 70 students) to Mercury: `TI=text and (AU=Gravano or ... or AU=Kao)` ∎

Although we illustrated the semi-join technique with an example where the query itself is a semi-join and the formulation of the query required a single invocation of the text system, these restrictions may not be necessary, as explained below.

Assume that $P(x_1, \ldots, x_k)$ is the set of foreign (text) join predicates in the `where` clause of a semi-join query, where each x_i is a join column of the relation. When the x_i's are instantiated, P reduces to a selection condition on the text retrieval system. Let the jth tuple of the joining relation have attribute values $(x_{1_j}, \ldots, x_{k_j})$ in the join columns. Then, the semi-join method asks the text system to evaluate the query $\bigvee_j P(x_{1_j}, \ldots, x_{k_j})$ (denoted by $\mathcal{Q}$). In contrast, the tuple substitution method makes an invocation $P(x_{1_j}, \ldots, x_{k_j})$ for each j. Note that if the number of search terms (say, $|\mathcal{Q}|$) in $\mathcal{Q}$ exceeds the bound M on the number of search terms allowed in a text query, then $|\mathcal{Q}|/M$ semi-join queries need to be sent. Finally, the semi-join technique generalizes to non semi-join queries in the case where the join predicates can be evaluated by relational text processing – upon fetching documents, we do subsequent relational text processing. We refer to this method as SJ+RTP.

3.3 Probing-based Techniques

When the foreign join between the text and the relation has multiple join predicates, then a novel set of techniques based on *probing* becomes applicable, as explained below.

The idea behind probing is to avoid sending queries to the text system with instantiations that return no matching documents. We will refer to such queries as *fail-queries*. For example, if while doing tuple substitution no matching documents are found for a tuple t in the relational table, the query obtained by instantiating with any tuple t' that agrees with the tuple t in the join columns will be a fail-query. Invocation of such fail-queries may be avoided by caching the information on results of past queries. Thus, in this case, invocation of a query for t' may be avoided if we kept the information that the query with t resulted in no matching documents. Probing offers significantly more opportunities for avoiding invocation of fail-queries.

Given a text-relational query Q, a *probe* on a set of columns P is a query Q_P obtained by removing all join predicates from Q except those on the set P, and requiring the text system to return only the information whether there are any matching documents. This may be accomplished by requesting the short form response from the text system. A column in the set P is called a *probing column*. It follows that if $Q_P(t)$ is unsatisfiable, then a query $Q(t')$ is unsatisfiable if $\pi_P(t) = \pi_P(t')$, i.e., if the tuples t and t' agree on the columns in P. Thus, *the failure of a probe to return any matching document tells us that instantiating the query with tuples that agree with the probe on the probing columns will also yield fail-queries*.

Example 3.4: In the following query, relation `project` stores information about research projects. The query selects names and members of NSF-funded projects, together with the docids of technical reports that have the project names in the titles and are written by the project members.

```
Q3: select project.member, project.name,
           mercury.docid
    from project, mercury
    where project.sponsor = 'NSF'
          and  project.name in mercury.title
          and  project.member in mercury.author;
```

Let us consider a probe `Q3'` for `Q3` with the probing column `project.name`. If there are no documents in Mercury with the word `PWS` in the title, then the probe `Q3'`, instantiated with `project.name = 'PWS'` returns *no* matching documents. Therefore, we can conclude that tuple substitutions with all tuples that have `project.name = 'PWS'` (say, with `project.member = 'De Smedt'` or `project.member = 'Pham'`) will return *no* matching documents. ∎

Intuitively, for probes to have a significant effect on reducing the invocation of fail-queries, we require (1) the predicates on the probing columns to be selective; and (2) the cardinality of the relation, projected over the probing columns, to be a relatively small fraction of the total number of tuples. This way, the failure of an instantiated probe helps avoid invocations of other fail-queries. These considerations lead to the conclusion that the choice of the set of probing columns crucially influences the effectiveness of the probe. The set of probing columns is determined at compilation time by the optimizer using a cost model. Section 5 discusses how the set of probing columns is selected.

Probing helps avoid invocation of failed queries and in itself is adequate for a semi-join of the relation with the text, i.e., if we want to project only over the attributes of the relation. We will use this capability in Section 6 to do multi-join query optimization. However, to implement a join, we need to use probing with other join methods. Probing can be used not only in conjunction with tuple substitution, but also with the relational text processing method.

Probing with Tuple Substitution:

For the following description, we assume that a cache is used to remember the set of all past probes (for one query), so that no duplicate probes are sent. For a given tuple t of the relation, the steps in Probing with Tuple Substitution (P+TS) join method are:

if cache has `fail` entry for probe of t **then** exit
  else
      Instantiate the query with t
      (as in tuple substitution)
      if cache has entry for probe of t **then** exit
        else update probe cache
      endif
endif

The step to update the cache is straightforward. If the query returned nonempty result set, then the entry for the probe is marked `success`. However, if the query returns no answers, then a probe needs to be sent. The cache is updated according to the result of the probe, i.e., marked `fail` if the probe fails and marked `success` otherwise. Note that we send a probe only *after* a given query has failed and also ensure that no duplicate probe is sent. Thus, the algorithm avoids generating redundant probes.

In case the tuples of the relation are ordered (or, grouped) by the probing columns, then no caching is required. Furthermore, in such a case, a probe is sent only if there is at least another tuple in the relation with the same values in the probing columns as the tuple which resulted in a fail-query.

Example 3.5: Consider the previous example. First, assume that the relation `project` is unordered. In such a case, we first do a tuple substitution for the tuple `project.name = 'PWS'` and `project.member = 'Desmedt'` and on failure, probe on the column `project.name`. Since the probe fails, no query is invoked for the tuple `project.name = 'PWS'` and `project.member = 'Pham'`. Next, assume that the relation `project` is ordered by `project.name` and that the project `'Foo'` has only one member `'Smith'`. Then, a failure on tuple substituting `project.name = 'Foo'` and `project.member = 'Smith'` does not lead to any probes. ∎

The cost model will be described in the following section, but here we touch upon a simple cost analysis to quantify possible benefits of the probing phase by contrasting it with the cost of simple tuple substitution. Assume that the cardinality of projection over J columns is N_J and the number of tuples in the relation is N. If s is the joint selectivity of predicates over the J columns, then the number of queries sent in this method is $N_J + sN$, whereas the number of queries sent by tuple substitution method is simply N. If $N_J < (1-s)N$, then the probing based tuple substitution sends fewer queries.

Probing with Relational Text Processing:
This method is similar to probing with tuple substitution. In this method (designated P+RTP), if a probe succeeds, then the documents matched by the probe are transmitted to the relational end where the remaining join predicates are evaluated using SQL capabilities.

As in the previous method, this technique also requires caching past invocations on the probe columns, i.e., whether the result was `success` or `fail` (or, the relation should be ordered on probing columns). However, in addition to the above information, we need to record the list of docids returned by the probe. We also maintain a cache of documents, the *doc-cache*, subject to a cache replacement policy. The doc-cache reduces the need to fetch documents repeatedly.

The following steps are performed for each tuple in the relation: (1) The cache is looked up to see whether there was a past invocation on the same values of the probe columns. (2) If so, then we check the doc-cache to identify documents that need to be retrieved (doc-cache miss). (3) Otherwise, a probe is sent and the list of docids are obtained and documents that need to be retrieved are identified.

For this method to be effective, the probe conditions must be selective so that not too many documents are retrieved from the text system and there must be significant overlap among documents needed in different invocations so that the total number of documents retrieved is small. Note that simple relational processing is dependent on the selectivity of the text selection conditions only and *cannot* take advantage of such overlaps. The following example illustrates an application.

Example 3.6: The following query selects students in the distributed systems area who have co-authored reports with their advisors.

```
Q4: select * student, mercury
    where student.area = 'distributed systems'
          and student.advisor in mercury.author
          and  student.name in mercury.author;
```

Suppose the number of distinct advisors is small and they are not very prolific. Then a small number of documents are retrieved, and the relational database system can quickly evaluate them against the tuples with successfully probes. ∎

4 Cost Analysis

In this section, we develop a cost model for computing the cost of each of the join methods discussed above. We first identify some basic cost parameters for accessing an external text source and relational text processing. We then look at what statistics are needed for the cost model and how to extract them. Finally we present the cost formulas for the join methods.

4.1 Basic Cost Constants

The cost involved in accessing a text retrieval system consists of three components: invocation cost, processing cost, and transmission cost. The invocation cost includes the overhead of invoking a foreign function and transmitting the query. We assume this is a constant cost c_i sec.

To process a query, current text systems typically read the inverted lists into main memory and perform set operations on these lists. As described in Section 2.1, the lists are sorted and therefore set operations take time linear in their lengths. We assume that the intermediate lists also fit in main memory, and there is no writing out of lists. Similar assumptions were made in [DH91] and were found to be consistent with the empirical data. The processing cost is thus proportional to the sum of the number of postings on the inverted lists processed and we denote the proportionality constant by c_r sec/posting.

The transmission cost includes the cost of transmitting the results. We assume the total cost is proportional to the cardinality of the result set. The transmission costs for long and short form documents differ and are represented by proportionality constants c_t^l and c_t^s (sec/document) respectively. We summarize this discussion with the cost formula for a single text search:

$$c_i + c_r \times \text{sum of lengths of inverted lists processed} + c_t \times \text{cardinality of result set}$$

We executed a number of queries and calibrated the integrated OpenODB-Mercury system. The values obtained are: $c_i = 3, c_r = 0.00001, c_t^s = 0.015, c_t^l = 4$. We note that (1) the invocation cost is significant – each requires a connection to the Mercury server; and (2) the long-form transmission cost is orders of magnitude more expensive than the short-form cost as each retrieval requires a separate connection also.

In our join methods, we may perform relational text processing by string matching. We assume the cost is proportional to the number of the documents and the proportionality constant is c_a sec/document.

D	total number of documents in the text database
M	maximum number of predicates in a text search
c_i	invocation cost const.
c_r	text system processing cost proportionality const.
c_t^s	short form transmission cost proportionality const.
c_t^l	long form transmission cost proportionality const.
c_a	relational text processing proportionality const.
N	number of joining tuples
k	number of join predicates
N_i	number of distinct values in column i
s_i	selectivity of predicate i
f_i	fanout of predicate i

Table 1: Cost Model Parameters

4.2 Predicate Selectivity and Fanout Estimation

As input to the cost model, the optimizer needs statistics extracted from the textual data. For a predicate i (say, column i `in` field i), two statistics are of interest: *predicate selectivity* s_i, defined as the probability that a term (which can be a word or a phrase) in column i occurs in field i of some document; and *predicate fanout* f_i, the average number of the documents in which a term in column i occurs in field i. The former is used in estimating the reduction effect of probing, and the latter in estimating the result-set size.

To estimate these statistics, we employ sampling techniques. We sample terms from column i, access the text retrieval system to check if they appear in field i of some document, and obtain the frequencies if so. The estimates thus obtained are maintained by the optimizer, and the sampling cost is amortized over queries with the same predicate.

Predicate Correlation

When there are more than one text join predicates, we need to estimate the joint selectivity as well as the joint fanout. For joint selectivity, if we assume that the terms in the join columns are correlated, i.e., frequently occurring together in the same documents, then the joint selectivity is equal to the minimum among them. On the other hand, if we assume that they are independently distributed in documents, then the joint selectivity should be the product of selectivities.

We generalize this as follows. Given a set K of k predicates with selectivities $s_1, \ldots, s_k$, the joint selectivity may depend on the g most selective predicates. If $s_{i_1} \leq \ldots \leq s_{i_k}$, $1 \leq i_l \leq k$, then the joint selectivity in a *g-correlated cost model* is $\mathcal{S}_{g,K} = \prod_{l=1}^{g} s_{i_l}$. For example, a 1-correlated cost model assumes full correlation and a k-correlated cost model for K assumes fully independent predicates. We can extend this to the joint fanout; if $f_{j_1} \leq \ldots \leq f_{j_k}$, $1 \leq j_l \leq k$, the joint fanout in a g-correlated cost model is $\mathcal{F}_{g,K} = \prod_{l=1}^{g} f_{j_l}/D^{g-1}$, where D is the total number of documents in the database.

4.3 Cost Formulas

We now present cost formulas for join methods that reflect the total resource usage. The cost formulas presented here reflect costs of joins with the text system. The formulas for text selections are omitted for space consideration. Since the cost of reading the relation is the same for all join methods, we omit that component of the cost. We also simplify the formulas by ignoring caching costs. We assume the joining relation has N tuples and there are k joining columns (the set of the joining columns is K): column 1 `in` field 1 `and` ... `and` column i `in` field i `and` ... `and` column k `in` field k.

We first derive some useful expressions. Suppose we send out n searches to the text system, each search being formed by tuple substitution into a subset J of the join columns. The total number of documents in the n result sets is $V_{n,J} = n \times \mathcal{F}_{g,J}$. Some of these documents may match more than one searches. The number of *distinct* documents matched is (assuming terms in different tuples occur independently in documents): $U_{n,J} = n \times D \times (1 - (1 - \mathcal{F}_{g,J}/D)^n)$. We are also interested in the sum of the lengths of the inverted lists retrieved for processing these searches. If we assume that a posting contains only a docid, and that the column width is one, then the sum of lengths is $\sum_{i \in J} f_i$. The formula for the general case is given in [CDY]. For n searches, the sum is $L_{n,a,b} = n \sum_{i \in J} f_i$.

The formula for TS follows from the above derivation:

$$C_{\mathrm{TS}} = c_i N + c_r L_{N,K} + c_t^s V_{N,K}.$$

Next we derive the formula for probing + tuple substitution. During probing, the total number of probes is equal to the number of distinct tuples in the projection of the probe columns. We denote this number by N_J, where J is the set of probe columns. For a single column probe, N_J is just the number of distinct values in the probe column. For multi-column probes, we estimate N_J by $\min(\prod_{i \in J} N_i, N)$, where N_i is the number of distinct values in column i. This is an overestimation and thus ensures that probing is favored only when the default method of tuple substitution is expected to perform significantly worse. Thus, the probe cost is estimated as: $C_{\mathrm{P}} = c_i N_J + c_r L_{N_J,J} + c_t^s V_{N_J,J}$. After probing, we do tuple substitution for tuples whose probes succeed. The number of such tuples is $R = N * \mathcal{S}_{g,J}$. So the total cost for probing + tuple substitution is

$$C_{\mathrm{P+TS}} = C_{\mathrm{P}} + c_i R + c_r L_{R,K} + c_t^s V_{R,K}.$$

The cost formulas of other methods are similarly derived [CDY]:

$$C_{\mathrm{P+RTP}} = C_{\mathrm{P}} + c_t^l U_{N_J,J} + c_a V_{R,J}$$
$$C_{\mathrm{SJ+RTP}} = c_i X + c_r L_{N,K} + c_t^l X U_{\frac{N}{X},K} + c_a \frac{N}{X} U_{\frac{N}{X},K}$$

where $X = \lceil Nk/M \rceil$.

5 Optimization of Single Join Queries

Optimization of queries that involve a single stored relation and the text retrieval system reduces to the problem of choosing among the join methods presented in Section 3 based on the above cost model. However, for probe-based methods, we must also determine an *optimal set of probe columns*. The rest of the section is devoted to addressing this question.

Optimal Set of Probe Columns:

We address the question of determining the optimal set of probe columns for the probe + tuple substitution join method. The corresponding algorithms for other probe-based join methods (e.g., relational processing) are given in [CDY].

Let us consider the case of single column probes. The optimal one-column probe can be determined in time linear in the number of join columns with the text retrieval system by iterating over the latter. However, note that since the cost of probing on a column depends on the number of unique values in that column as well as on the selectivities of the predicates on that column, the optimal single column is not necessarily the column which has predicates with minimal join selectivity, as illustrated in the following example:

Example 5.1 : For simplicity, assume that we are considering text systems where the cost of invocation dominates significantly. Thus, $c_r = c_t = 0$. In such a model, the cost of probe + substitution on ith column is $N_i + s_i N$. Thus, even if $s_i > s_j$, the probe on ith column may still be better than the probe on column j if $N_i + s_i N < N_j + s_j N$. In other words, $s_i - s_j < (N_j - N_i)/N$. ∎

As we increase the number of probing columns, more join predicates are evaluated as part of the probing phase. The joint join selectivity of these predicates is lower than predicates on one of the columns. On the other hand, the number of distinct values in the group of probing columns (i.e., the parameter N_J in the cost formulas) increase when the number probing columns in J are increased. Because of these two opposing factors, increasing the number of probing columns may or may not lower the cost of probing + tuple substitution. The following example shows that a two column join may dominate a single column join.

Example 5.2: Consider the cost model where joint selectivity is given by the product of the individual selectivities. For simplicity, consider only the invocation cost. Assume that there are three join predicates on columns 1, 2 and 3. Note that N denotes the number of tuples in the relation and $s_2 = s_3$.

$$N = 10^7, N_1 = 10^3, N_2 = N_3 = 10, s_1 = .005, s_2 = .01$$

It may be seen that N_1 is the best choice for the single column case. However, $\{N_1, N_2\}$ is the best 2 column probe and is a better choice compared to the 1-column probe on N_1. ∎

In general, if there are k join predicates with text retrieval system, in the worst case each predicate may be on a distinct column. Therefore, there could be as many as 2^k subsets of columns each of which qualifies as the optimal set of probing columns. Thus, choosing among them may take $O(2^k)$ time. However, the complexity of determining the optimal set of probe columns for 1-correlated models of joint selectivity (See Section 4) is surprisingly low. As the following theorem shows, for such cost models, we need at most 2-column probes. Therefore, determining the optimal set of probe columns take $O(k^2)$ time only.

Theorem 5.3 : *For 1-correlated cost models, the optimal set of probing columns has at most 2 columns.*

Proof Sketch: The proof is by induction and we show that if an i-column probe S_i is optimal where $i > 2$, then there must be a subset $S_{i-1} \subset S_i$ such that the $(i-1)$ column probe on the set of columns S_{i-1} is optimal as well. In the rest of this proof, we show that (1) an 1-column probe suffices if only invocation cost is measured. (2) an 1-column probe suffices if only processing cost is measured. (3) a 2-column probe suffices if only transmission cost is measured. The proof for the general case, where all three costs must be accounted for, follows from (1)-(3). ∎

The above theorem generalizes when we consider g-correlated cost models and we can show that the number of columns in the optimal probe is no more than $Min(k, 2g)$. In the presence of an existing order on the join columns of the relation, we compare the cost of optimal probe, determined as above with the cost of probing on the ordered columns. Note that the latter incurs no cache lookup or update cost.

Finally, note that although probe, followed by relational text processing is an attractive join method, it suffers from the danger that if the selectivity and fanout estimates are unreliable, then too many documents are fetched. We rely on runtime optimization techniques to address such difficulties [CDY].

6 Optimization of Multi-Join Queries

In this section, we consider extensions to the execution space that result if the query involves joins with more than one relation.

Traditional Execution Space:

The execution of a query is traditionally represented syntactically as *linear join trees* where the internal node is a join operation and each leaf node is a base table. The annotations on the nodes provide details such as selection conditions, choice of access paths and join algorithms. The set of all annotated join trees for a query that are considered by the optimizer, is traditionally called the *execution space* of the query. The aim of the optimizer is to produce an execution plan of least cost from a given execution space. Traditionally, the execution space has been limited to left-deep join trees which correspond to linear orderings of joins. The optimality of a plan is with respect to a cost model. We assume that the cost model satisfies the *principle of optimality* which enables use of a dynamic-programming based solution, described below.

Traditional Enumeration Algorithm:

The well-known algorithm [SAC+79] to choose an optimal linear ordering of joins in a relational query uses dynamic programming. A query Q is viewed as a *set* of relations (say, $\{R_1..R_n\}$) which are *sequenced* during optimization to yield a join order. The optimization algorithm proceeds stage-wise, producing optimal plans for subqueries in each stage. Thus, for Q, at the ith stage ($2 \leq i \leq n$), the optimizer produces optimal plans for all subqueries of Q of *size* i (i.e., subqueries that consist of join of i relations). Consider a subquery Q' of size $i+1$. The steps in the function *Enumerate* are followed to find its optimal plan.

Function *Enumerate*

1. **for all** R_j, S_j s.t $Q' = S_j \cup \{R_j\}$ **do**
2. $\quad p_j := joinPlan(optPlan(S_j),\ R_j)$
3. **end for**
4. $optPlan(Q') := MinCost_j(p_j)$

In steps 1 and 2, all possible ways in which a plan for Q' can be constructed by extending an optimal plan for a subquery of size i (i.e., S_j) are considered. Note that R_j is the remaining relation that occurs in Q'. The function *joinPlan* creates the plan for joining a relation with another intermediate (or a base) relation. Thus, access methods and choice of join algorithms are considered in *joinPlan*. In step 4, $MinCost$ compares the plans constructed for Q' and picks a plan with the least cost. It can be shown that the above algorithm is optimal with respect to the execution space of left-deep trees. The complexity of the enumeration can be characterized in terms of number of 2-way join optimization tasks that are invoked by the enumerator. With such a measure, it can be shown that if the query is a join of n relations (including the text source), then the complexity of enumeration is $O(n2^{n-1})$. We refer the reader to [SAC+79, GHK92] for more details.

New Execution Space:

The simplest approach is to treat the foreign join no differently from any other relational join from the perspective of execution space, i.e., assume that in the left-deep trees, one of the join nodes could be a join with the text system. In such a case, we need no changes to the enumeration algorithm outlined above. However, if we treat a foreign join as a relational join, then all join and selection predicates on the text retrieval system are *evaluated together*. Such an evaluation strategy is appropriate for relational systems since once a tuple has been obtained by index access or as a result of a join, the remaining selection and join predicates can be evaluated with no I/O cost. The situation in case of accesses to text databases is quite different. First, evaluation of a join predicate between the text database and the relational table requires that the joining column on the relational side is instantiated. Therefore, by requiring that we evaluate all join predicates together, we must delay the join until all join predicates are evaluable. However, the following example shows that the above scheme may not be optimal.

Example 6.1 : Consider the following query which identifies documents published in 1993 and coauthored by a student and a faculty from another department.

```
Q5: Select student.name, mercury.docid
From student, faculty, docid
Where student.name in mercury.author
and   faculty.name in mercury.author
and   faculty.dept != student.dept
and   'May 1993' in mercury.year
```

If we want that all join predicates with mercury are evaluated together, then the join between `student` and `faculty` must precede the join with Mercury in a left-deep execution tree. However, the only join predicate in such a case is `faculty.dept != student.dept`, which is expected to be of low selectivity. On the other hand, few of the students write articles. Therefore, the predicate `student.name in mercury.author` has low selectivity. Therefore, if we evaluate the above predicate before the join with `faculty`, it reduces the size of the `student` relation, thereby lowering the cost of the join between `student` and `faculty`. The evaluation of the predicate `student.name in mercury.author` can be achieved by sending a probe from `student` to `mercury`. Finally, we evaluate the remaining join predicate with the text system. ▮

Probe as a Semi-join:

In the above scheme, the use of probe as a semijoin of the relation `student` by `mercury` helped reduce the cost of the join between `student` and `mercury`. In turn, this also helped reduce the cost of evaluating the remaining join predicate on the text system since the join of the reduced student relation with mercury is significantly less than the cost of the join of the full `student` relation with `mercury`.

In view of our observation, we define the execution space of the optimizer to contain the set of all *PrL trees*, defined as follows: (1) A left-deep tree is a PrL tree (2) Every left-deep tree which is augmented with additional probe nodes placed between two relational join nodes or between a scan node (for a relation) and a relational join node is a PrL tree. The probe nodes must precede the join node with the text system.

Unlike left-deep trees, PrL trees allow for the possibility of reducing relations by using the selectivity of the text join predicates. Thus, in Example 6.1, the execution where there was a probe node between the scan node for `student` and the join node between `student` and `faculty`, is an PrL tree but not a left-deep tree. Note that such probe nodes can only precede the join node with the text since any probes following the text join node will be redundant.

The execution space of PrL trees is significantly larger than that of left-deep trees. First, for each probe node, we need to determine the predicates which should be used for probing. Interestingly, although probes achieve the effect of semi-join, determining the optimal probing columns is unique to foreign joins and was a non-issue in traditional semi-join processing since evaluation of additional join predicates and selection conditions was free. Since evaluation of each join predicate requires a distinct index access (and retrieval of the set of docids), it may not be desirable to probe on all possible predicates. Next, we observe that we cannot compare two subplans over the same set of relations, one with application of a probe and the other without its application, simply by cost. This is due to the fact that application of a probe may reduce the size of the relation, even if it increases its cost. Thus, the traditional plan for $\{R_1, R_2\}$ cannot be compared by cost with the plan corresponding to the PrL tree where a probe is applied after the scan of R_1 and then a join is taken with R_2, as the latter may result in a reduced relation compared to $R_1 \bowtie R_2$. Thus, there will *not* be a *single* optimal plan for $\{R_1, R_2\}$.

Modified Join Enumeration Algorithm:

The solution that we present is guided by the following two desiderata (1) The cost of the plan picked by the optimizer should be *no worse* than the plan obtained in the traditional execution space (2) The increase in the cost of optimization must be moderate - a requirement that is particularly important for interactive queries. The solution that we propose satisfies both these criteria. Moreover, the solution requires little adaptation to the existing enumeration algorithm.

First, given n relational tables and the text system, the optimizer enumerates all possible orderings of the relations and the text system. Intuitively, the positioning of the text system in the join order indicates where all the join predicates are evaluated and the necessary text information is obtained from the text system. Observe that by our definition of PrL trees, all probe nodes (used as semi-joins) must precede the position of the next join. Next, at each step of the outer loop in the function *Enumerate*), we consider the following four alternatives in Step 2 and choose the plan which has the least cost. Note that some of the plans in (a)-(d) may not be well-defined if R_j or S_j does not have any predicate with the text system:

(a) $joinPlan(optPlan(S_j),\ R_j)$
(b) $joinPlan(probe(optPlan(S_j)),\ R_j)$
(c) $joinPlan(optPlan(S_j),\ probe(R_j))$
(d) $joinPlan(probe(optPlan(S_j)),\ probe(R_j))$

Example 6.2: Let us consider the optimization of the query `Q` in the previous example. While determining the optimal plan for $\{$`student`, `faculty`$\}$, the optimizer also considers the costs of $\{$`student'`, `faculty`$\}$, $\{$`student`, `faculty'`$\}$, as well as $\{$`student'`, `faculty'`$\}$, where `student'` and `faculty'` designates relations reduced by probes. Note that while the size of $\{$`student'`, `faculty'`$\}$ is the least, the plan may be more expensive than $\{$`student`, `faculty`$\}$ due to the presence of two probe operators. ∎

In each of the above alternatives, we must also determine the probing columns that are optimal to reduce the cost of *joinPlan*. Thus, we use the cost formulas for probe from S_j, followed by join with R_j to determine the optimal probe columns. The analysis takes the same form as in Section 5.

Note that the above enumeration algorithm ensures the set of all traditional left-deep trees are enumerated as well. Therefore, the algorithm *never* picks a plan that is worse than its traditional counterpart. In terms of optimization cost, considering probes is analogous to considering additional access methods. Therefore, the asymptotic complexity of optimization is bounded by $O(n2^{n-1})$, same as in the traditional enumeration. Other choices of the execution space as well as a detailed comparison of the modified enumeration algorithm with the traditional one is discussed in [CDY].

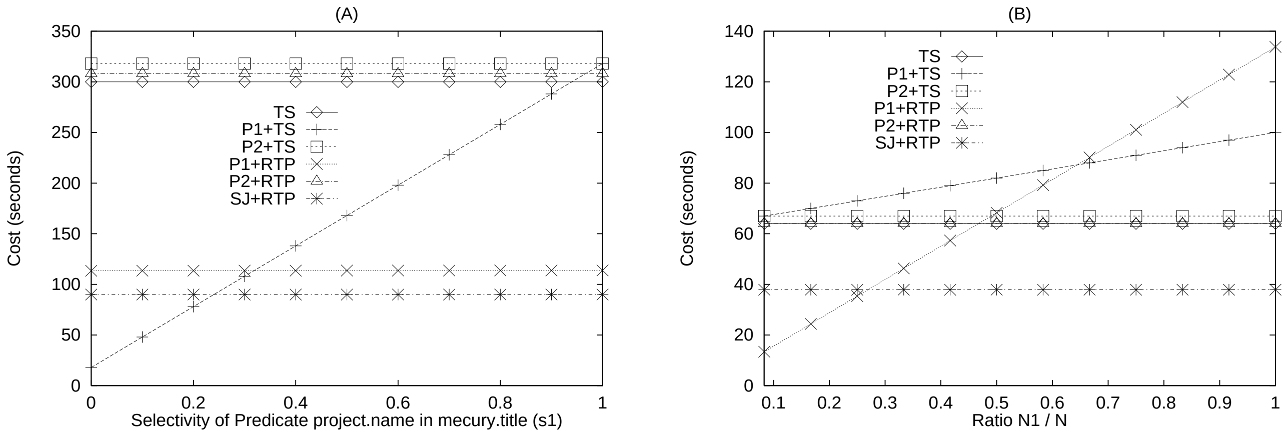

Figure 1: (A) Varying s_1 in Q3 (B) Varying N_1/N in Q4

7 Experimental Results

To establish empirically that each join method is optimal in certain settings, we executed the queries in the examples discussed above (Q1-4) in the integrated OpenODB-Mercury system. Recall that Mercury stores computer science technical reports from a number of universities. On the database end, we created a relational database that models a university computer science department. For each query, we implemented the various methods in OSQL, the database programming language for OpenODB. For the TS method, we assumed the variant in which only distinct tuples in the joining relation are sent out. We measured the time each method took to execute; averages over repeated runs are reported.

Join Method	Q1	Q2	Q3	Q4
TS	145	52	328	43
RTP	**8**	91	–	–
SJ+RTP	18	**9**	97	20
P+TS	–	–	**81**	52
P+RTP	–	–	118	**12**

Table 2: Execution Times for Sample Queries

We verified that our cost formulas in Section 6 correctly predict the optimal method for each query, using the fully correlated cost model.

7.1 Varying the Parameters

In this section, we study how the choice of methods changes as the characteristics of the relational data vary. We started with the parameter setting of a query above, and varied certain parameters (s_i's, N_i's, and N) in turn over a range of values. For each value, we used the cost formulas to compute the costs of the methods.

We repeated the experiments for each multiple-column join queries above (Q3 and Q4). Due to space consideration, we only show some of the interesting graphs below. We use the notation Pi+TS below to refer to the Probe + Tuple Substitution method where the probe is on column i, and similarly for Pi+RTP.

Figure 1(A) shows the variation in the costs of the methods as s_1 changes from 0 to 1. Parameter s_1 represents the percentage of the first (`project.name`) column values that are found in the text field `title` of some document. For each value of s_1, the optimal method is the one with the lowest cost. The original value of s_1 in Q3 is 0.16. Comparing the actual execution times in Table 2 with the computed cost at $s_1 = 0.16$ in Figure 1(A), we see that our cost model predicts the ranking of the methods. When s_1 is increased, more and more probes succeed and thus P1+TS sends off more and more text searches and becomes more expensive. Thus P1+TS becomes more expensive and SJ+RTP is the optimal plan.

Next we consider Q4 and vary the ratio N_1/N, the number of distinct values in the `advisor` column over the total relation size. The results are shown in Figure 1(B). Varying this ratio impacts the methods that probe on column 1. For P1+TS, as N_1/N increases, more probes result and all of them succeed (s_1 is fixed at 1), and so the number of text searches increases. Similarly for P1+RTP, more and more probes are sent out. The total number of documents matched by the probe column increases as N_1/N increases and f_1 is kept fixed. Consequently many more documents are shipped to the relational side, resulting in the rise of the cost of P1+RTP.

We verified this by re-instantiating the relation `student` with $N_1/N = 1$ and executed Q4 again in the integrated system. The ranking of the execution times was as predicted by the cost formulas.

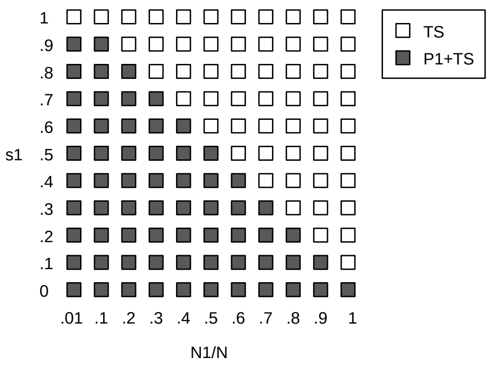

Figure 2: TS vs. P+TS for Q3

7.2 P+TS vs. TS

The results above indicate that the relational text processing methods (especially SJ+RTP) can be quite efficient when SQL-supported string processing is applicable. However, in general, only two methods are universally applicable: TS and P+TS. In this subsection, we focus on these two methods, and compare their performances.

First we compare TS with P1+TS in Q3. Two parameters are important in determining their superiority: s_1, the selectivity of the probing column, and N_1/N, the ratio between the number of distinct values in the probing column and the relation size. We thus vary s_1 from 0 to 1, and N_1/N from 0.01 to 1 ($N = 100$ in Q3), covering all possible scenarios. For each combination of s_1 and N_1/N, we compute the costs using the cost formulas. (Other parameters are kept at original values.) The winning method at each point is marked in Figure 2.

We can see that each method constitutes about half of the space. We repeated the same experiment with Q4 and obtained similar results, with TS taking slightly more space than P1+TS. The results can be explained as follows. The cost of accesses to Mercury is dominated by invocation and transmission components. As the number of long-form documents transmitted is the same for both methods, we may focus on invocations only. The number of invocations in TS is simply N, while that in P1+TS is $N_1 + s_1 N$. The area occupied by P1+TS should thus be $N_1 + s_1 N < N$, or $s_1 < 1 - N_1/N$, which is approximately the area shown in Figure 2.

8 Discussion

In this paper we have presented a set of join methods to retrieve information from text systems. When there are multiple join predicates, *probing* based join techniques are valuable and we have shown how to pick an optimal set of predicates for the probe. Our experiments confirm the importance of the join methods that we have proposed. A new definition for extended execution space as well as an enumeration algorithm to search that space was presented. The search technique that we have proposed can be implemented with few changes to existing optimizers and does not pay significant overhead in optimization time. Although we have presented a simplified cost model in this paper, the techniques we proposed are robust enough to apply more generally.

The model of text system that we have assumed is typical of commercial systems available today. Could such systems be engineered differently to facilitate loose integration? We observe that the text system can help the optimizer by making available statistics such as distribution of fanout of the words in the vocabulary. Such information will eliminate the need for sending all single-column probes to the text system. Furthermore, if text systems provide the ability to accept multiple queries in one invocation and can return answers in a batched mode while maintaining the correspondence between each query and its answers, then (as in the case for semi-join) invocation and possibly transmission costs for the queries will be reduced.

Another natural question is whether our techniques can be used for text systems that are based on other retrieval models (e.g., vector-space, probabilistic). We observe that the semantics of predicates is different in these models from that in Boolean text systems and relational systems. In particular, adding predicates in a query in these text systems may result in more answers. In contrast, our techniques rely on the traditional semantics of predicates. Thus, processing and optimization of join queries between structured database and these models require addressing semantic issues that we have not addressed here. Therefore, our techniques will *not* be directly applicable in such systems.

Beyond Text Systems: We now consider whether the techniques that we have proposed for integration with text systems generalize to other external data managers. The semi-join technique applies whenever the query interface to the foreign system allows specification of a relatively *large set of disjuncts*. Methods such as relational text processing rely on the ability of the relational systems to support *methods* (with consistent semantics) available in the foreign system (in this case, it was the ability to do substring matching). The join methods based on probing rely on the fact that each predicate on the foreign system must be evaluated by index lookup which is true of storage systems for image and other multimedia objects as well. Likewise, the conclusions about the execution space for multi-join queries apply whenever evaluation of predicates is through indexes only. Thus, the techniques presented in this paper apply to a broader class of foreign systems beyond Boolean text systems. While our experiments confirm the importance of these techniques for text, their applications to other domains need to be studied.

References

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *VLDB*, 1993.

[AS91] W. Aref and H. Samet. Optimization strategies for spatial query processing. In *VLDB*, 1991.

[BG92] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geomtric database system. *ACM Transactions on Database Systems*, pages 247–303, June 1992.

[BGWR81] P. A. Bernstein, N. Goodman, E. Wong, and C. Reeve. Query processing in a system for distributed databases. *ACM Transactions on Database Systems*, 6(4), 1981.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. Query processing in a multimedia document system. *ACM Transactions on Office Information Systems*, 6(1), 1988.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *SIGMOD*, 1994.

[CDY] S. Chaudhuri, U. Dayal, and T. Yan. Text and databases: Query processing and optimization. *HP Labs Technical Report*, in preparation.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *VLDB*, 1989.

[CHK+91] T. Connors, W. Hasan, C. Kolovson, M. A. Neimat, D. Schneider, and K. Wilkinson. The papyrus integrated data server. In *Proceedings of PDIS*, 1991.

[CM94] M. P. Consens and T. Milo. Optimizing queries on files. In *SIGMOD*, 1994.

[CMU94] CMU. *Welcome to Project Mercury*, 1994. URL http://rose.mercury.acs.cmu.edu.

[Cor94] Cornell University. *The CSTR Project*, 1994. URL http://cs-tr.cs.cornell.edu/Info/cstr.html.

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.

[DH91] S. DeFazio and J. Hull. Towards servicing textual database transactions on symmetric shared memory multiprocessors. In *HPTS*, 1991.

[Fal85] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):50–74, 1985.

[Fal92] C. Faloutsos. Signature files. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval Data Structures and Algorithms*, pages 44–65. Prentice Hall, 1992.

[GD87] G. Graefe and D. J. DeWitt. The exodus optimizer generator. In *SIGMOD*, 1987.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, 1992.

[Hew92] Hewlett-Packard. *OpenODB Reference Manual B3185A*, 1992.

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proceedings of the 1989 ACM-SIGMOD Conference on the Manage ment of Data*, pages 377–388, Portland, OR, June 1989.

[HHK+93] W. Hasan, M. Heytens, C. Kolovson, M. A. Neimat, S. Potamianos, and D. Schneider. Papyrus gis demonstration. In *SIGMOD*, 1993.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, Washington D. C., May 1993.

[KMP93] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *VLDB*, 1993.

[Lib94] Library of Congress. *Library of Congress World-Wide Web*, 1994. URL http://lcweb.loc.gov/homepage/lchp.html.

[LMH+85] G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms. Query processing in R*. In W. Kim, D. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*. Springer Verlac, 1985.

[LS88] C.A. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *VLDB*, 1988.

[LW90] W. Lee and D. Woelk. Integration of text search with orion. *IEEE Data Engineering Bulletin*, 13(1), 1990.

[MDZ93] G. Mitchell, U. Dayal, and S. B. Zdonik. Control of an extensible query optimizer: A planning-based approach. In *VLDB*, 1993.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[Sal89] G. Salton. *Automatic Text Processing*. Addison Wesley, Reading, Massachusetts, 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *SIGMOD*, 1990.

[YA94] T.W. Yan and J. Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *VLDB*, 1994.

[YC85] C.T. Yu and C.C. Chang. Distributed Query Processing. In W. Kim, D. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*. Springer Verlac, 1985.