Integrating SQL Databases with Content-Specific Search Engines.

Conference Paper · January 1997

Source: DBLP

CITATIONS
50

READS
182

2 authors, including:

Stefan Deßloch
Technische Universität Kaiserslautern
71 PUBLICATIONS **413** CITATIONS

SEE PROFILE

Integrating SQL Databases with Content-specific Search Engines

Stefan Dessloch, Nelson Mattos
IBM Database Technology Institute, San Jose, CA
dessloch@almaden.ibm.com, mattos@vnet.ibm.com

Abstract

In recent years, database research and product development activities have focused on support for non-traditional data types, such as text or multi-media documents. This paper describes an approach of coupling SQL databases and content-specific search engines, such as full-text retrieval engines, in an efficient manner. It is based on a query rewrite scheme that exploits so-called table functions, which are used to pass results from external search engines into the database engine. Using this approach the content-specific indexing mechanisms of search engines can be exploited without having to extend the database engine with new access methods, or having to break up the search engine to map its indexing scheme to database index structures.

1 Introduction

In recent years, database research and product development activities in the areas of object-oriented, extensible, and object-relational databases have focused on support for non-traditional data types, such as text or multi-media documents [Car86, CD96, Cha96, Kim95, Loh91, Schw86, Sto96, ZM90]. These activities have resulted in systems that support extensibility in terms of their type systems and their query languages. Such extensibility features permit the creation of new data types and new functions (or methods) to accommodate new types of content in the database as well as well as to manipulate and search such content.

**Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997**

Existing object-relational database products, such as IBMs DB2 Universal Database [Cha96, Dav96a] or Informix Universal Server [Dav96b] provide an architecture and APIs for integrating content management and search for new data types in the form of plug-ins.[1] This is especially attractive for vendors of content-specific search engines, giving them the opportunity to plug their existing search engine products into the database engine with minimal migration efforts, thereby providing database users with their advanced content search capabilities inside SQL.

This trend is also reflected in current standardization efforts, such as SQL3 [Mel96], ODMG [Cat96], and SQL/MM [Cot96a]. The current version of the SQL3 standard draft, which is expected to become an official standard in 1998, specifies language extensions for creating complex, user-defined types (abstract data types, or ADTs) and user-defined functions (UDFs). The SQL3 standard specification is 'supplemented' by the SQL/MM documents, which attempt to standardize the structure and behavior of multi-media data types such as text or image, as well as other non-traditional data types such as geospatial data, in the context of the SQL language.

In order to efficiently support the addition of such new data types, a database engine has to be truly extensible, meaning that the engine and the optimizer have to be able to recognize and execute user-defined types and functions in the same way as built-in ones. Moreover, indexing support has to be extended in such a way that it also covers user-defined data types.

Various approaches for extending indexing support have been developed over the last years [Aok91, DDSS95, GFHR96, LS88, Sto86]. In the relational, or object-relational context, some of these approaches permit the user or database administrator to create indices not only on table columns, but also on the results of function invocations or expressions involving columns. While indexing on the results of function invocations is definitely an important concept, it does not help at all in the support of content search on data types such as text documents or images. Content search on such data types usually involves predicates for specifying that a certain value or document should match a (potentially very complex)

1. IBM calls such plug-ins 'relational extenders' [IBM95], Informix calls them 'DataBlade ®'

search pattern. Neither storing the complete document nor storing the result of a simple function invocation on a document in an index helps to support such types of queries.
Other approaches have concentrated on developing new database access methods for non-traditional content, such as for text or spatial data [Gut84, Jag90]. While the incorporation of new access methods into the database engine is probably the most effective way to enhance indexing capabilities, it is also the most expensive one. In general, this is hardly possible by just plugging in some code into the engine. Because of its interaction with central database components such as locking and recovery management, adding an access method is a complicated task that requires advanced database skills and intimate knowledge of the underlying database system. Especially for a vendor that is specialized in search technology for specific types of content and would like to implement a database 'plug-in' for existing object-relational DBMS, adding an access method to the DBS may not be feasible. In addition to the above described database aspects, such a vendor also has to map his content-specific index data structures and the processing model of the indexing engine to the indexing approach of the database system. Thus, it is likely that because it is exactly the (usually proprietary) indexing and search technology where vendors differ and compete, a single access method provided by the database vendor for a specific type of content, such as d-trees for text, does not meet the requirements of such a vendor.
This paper describes an approach of coupling SQL databases and content-specific search engines, such as full-text retrieval engines, in an efficient manner. It is based on a query rewrite scheme that exploits so-called table functions, which are used to pass results from external search engines into the database engine. Using this approach the content-specific indexing mechanisms of search engines can be exploited without having to extend the database engine with new access methods, or having to break up the search engine to map its indexing scheme to database index structures.
The paper is organized as follows. In Section 2, we introduce a sample scenario from the area of text databases, which serves as a running example throughout the text, and illustrates the problem we are addressing with our proposed approach. In Section 3, we introduce the usage of table functions, which resembles a first step towards a solution. Section 4 describes the definition of index functions and the associated query rewrite approach. Additional aspects related to the index function rewrite are discussed in Section 5. A discussion of related work is presented in Section 6, and Section 7 resembles our final conclusions.

2 Text Search in SQL Databases - a Sample Scenario

In order to prepare the grounds for the discussion of our approach, we describe a sample scenario in the area of text search in SQL databases.

2.1 SQL3 and SQL/MM Full-Text

SQL3 defines statements for adding so-called abstract data types (ADTs) to the database type system. ADTs have attributes, whose values can again be ADTs. Moreover, subtyping with inheritance as well as encapsulation are supported. ADTs 'live' inside columns of relational tables, and they can only be inspected and modified using their functional interface, i.e., the set of functions defined for the ADT. Using the SQL mechanisms for creating ADTs and user-defined functions, any database user having the required privileges can extend SQL to accomodate, modify, and search over new types of content.
The SQL/MM Full-Text document [Cot96b] attempts to standardize the integration of text retrieval in SQL based on the concepts introduced in SQL3 by providing definitions of text-related ADTs and their functional interfaces. For example, the ADT 'FullText' is defined, together with a number of functions operating on the 'FullText' ADT. Once this ADT is available in a database, a user can define a table with a full-text column in the following way to create a table with information about projects.

```
CREATE TABLE projects (
    proj_no      integer,
    title        varchar(50),
    budget       integer,
    description  FullText)
```

In addition, SQL/MM defines a number of functions to work with values of type 'FullText', such as constructor functions and search functions. For example, the function 'contains' can be used to perform text search on columns of type 'FullText' in the following way.

```
SELECT proj_no, title
FROM compschema.projects
WHERE contains(description,
    ' "database" IN SAME SENTENCE AS
     "object-relational" ')
```

The above query would return all projects with a textual description that contains the word "database" in the same sentence as "object-relational". The 'contains' function involved in this query has two arguments (a value of type FullText and a search pattern string), and it returns a boolean value.
A vendor implementing the SQL/MM Full-Text specification can supply function libraries and DDL statements that can be executed to create the ADTs and additional functions in the scope of a database, so that a database user can use them in the above described manner.

2.2 Problem Description

It is obvious that the above SQL query involving text search cannot be efficiently executed without some sort of indexing scheme on FullText documents. Otherwise, the 'contains' function would have to be performed on the description column for each tuple in the projects table. This would dramatically impact the overall execution cost of the statement not just because of the full table scan

involved in the evaluation, but also because of the costs of evaluating the 'contains' function itself. Without appropriate index support, 'contains' would have to fully analyze the given document to determine whether it matches the search expression.

This type of search and indexing support is well-understood and commercially available in the area of information retrieval and full-text search engines. Such engines utilize index structures based on inverted word lists [Sal89], and they typically support APIs to

- construct a (named) index for a collection of documents in a given scope somehow identified by the user, and
- search for all identifiers of documents in a certain scope (given by the index name) that match a given text search pattern.

This is exactly the functionality required for index support in database-oriented text search, where the scope of an index is usually the FullText column of a base table, and FullText documents can be identified in the context of this table either by a row or tuple identifier, or by a unique key value (e.g., the primary key).

As discussed previously, there are various reasons why existing approaches, such as using indices on functional expressions, utilizing existing access methods, or adding a new access method to the database engine either do not apply, or are not desirable, essentially because they require to break up the existing text search engine and map its indexing and search scheme to the one of the database system. This forces the vendor to expose proprietary key technology by making it 'public' in the database index, which is any many cases not tolerable. We are therefore looking for a more light-weight approach that preserves the text search engine, and allows to utilize its index-based search technology inside of the database engine through its standard APIs. In such an approach, user-defined database functions (e.g., 'contains') would be realized in an external programming language (such as C), and would utilize the external search engine through its standard programming APIs.

2.3 Implementing SQL/MM Full-Text: the DB2 Text Extender

As an example of an implementation of the SQL/MM Full-Text specification, we introduce the DB2 Text Extender, a 'plug-in' developed for IBMs DB2 Universal Database product [Cha96, Dav96a]. DB2 Universal Database supports some of the object-relational features specified in SQL3, such as user-defined distinct types and user-defined functions. These features serve as a basis for the implementation of the Text Extender, which integrates text search into SQL by utilizing an IBM stand-alone text search engine called SearchManager.

Using Text Extender, the SQL statements introduced in Section 2.1 look slightly different. The projects table would now be defined in the following way.

```
CREATE TABLE projects (
    proj_no         integer,
    title           varchar(50),
    budget          integer,
    description     CHARACTER LARGE OBJECT,
    description_id  db2texth)
```

Text content is stored in the table using the traditional data types available for character data, such as variable length character data types, or character large objects. Each text column is 'accompanied' by an additional column of type 'db2texth', which is a user-defined distinct type introduced by Text Extender. The values of these columns (also called text handles) serve, among other things, to uniquely identify the text documents in the text column for the search engine. When issuing text search queries, the accompanying handle column has to be used instead of the text columns itself, as illustrated in the following query.[1]

```
SELECT proj_no, title
FROM compschema.projects
WHERE contains(description_id,
    ' "database" IN SAME SENTENCE AS
     "object-relational" ')
```

Figure 1 explains the basic architecture of the text extender in terms of the interaction of the database engine with the text search engine, and helps to illustrate how the above query would be evaluated. Please note that the architecture depicted in the figure is in so far incomplete as it illustrates only aspects related to text search UDFs. Additional functionality and components of Text Extender, such as client components and administrative APIs are not described. The query would be submitted via a DB2 client to the DB2 server engine. For each row in the projects table, the engine calls the contains function with the contents of the description_id column and the search pattern as arguments. The contains function is a user-defined function written in a 3GL (C). It again calls the text search engine (realized as a set of C functions in a shared library), passing it the search pattern as well as the name of the index covering the text documents stored in the description column of the projects table.[2] The text search engine returns the result of the text search to the contains UDF body in form of a list of document identifiers (i.e., values of type db2texth). The UDF checks whether the identifier that has been supplied by the database engine is actually contained in the result list returned by the text search engine, and returns the appropriate result ('true' or 'false') to the database engine. Based on the result of the contains function call, the database engine will construct the query result. It is possible for the contains UDF to keep the results returned by the text search engine across invocations inside of a query by using a special 'scratchpad' memory area supplied by the engine. Therefore, the actual text search using the external text search engine has to be performed only once, during the

1. This approach was chosen to be able to perform text search over existing character data columns without breaking existing applications. The ADT support will be added in a future release.
2. This information is stored in text extender system catalog tables in the database.

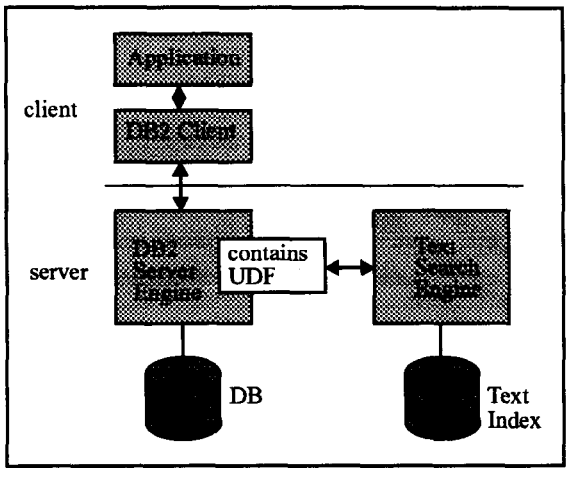

Figure 1: Interaction of database and text search engine in Text Extender.

first invocation of the contains UDF inside of a query. There are several issues related to this type of coupling that we would like to address very briefly.

- Text index creation is initiated using a separate text extender API implemented on top of DB2. It utilizes DB2 stored procedures to have the text engine build the index, which consists of a set of files stored outside of DB2 in an index directory on the server system.
- Meta information relevant for text search, such as the names of database columns enabled for text search and the names of the external text indices covering a certain database column are stored in additional 'catalog' tables. The catalog table is a regular DB2 table, whose contents are manipulated by the text extender admin functions, but can be read by the end-user using standard SQL.
- For constructing the external text index, a separate process is initiated that accesses the text catalog table and determines the text column containing the text data, as well as the associated handle column. It then reads the text data plus the handle on a per-document basis, and analyzes the text for constructing the index entries, storing the text handle value as a unique identifier of the document inside the index.
- DB2 triggers, which may also call UDFs in the conditions and trigger bodies, are used to reflect updates performed on text columns correctly in the corresponding text indices.
- The contains UDF for text search runs in the same process and address space as the database engine. This extremely minimizes UDF overhead by eliminating expensive inter-process communication.[1]

For a detailed description, the reader is referred to [IBM95, IBM96].

Problem Description Rephrased

Although the above described architecture permits the exploitation of an external text search engine through its native APIs, and even limits the interaction to a single call (performed during the first call of the contains UDF), there is no way to avoid a full table scan on the projects table. In other words, although the text search engine can provide the result in form of a set of identifiers in one call, the database engine will call the contains function for each row in the table. The main problem is therefore: How can the set of identifiers returned by a text index lookup (involving the API of the external text search engine) be fed back into the database query evaluation process in a way that is comparable to a 'traditional' database index lookup to avoid the table scan. Moreover, the mechanism to achieve this integration of results needs to be externally available through a database API, so that it can be utilized by anybody that wants to integrate support for new data types.

3 Using Table Functions - a First Step

Essentially, the problem described above lies in the mismatch of the text search engine access (one access per base table column or index, returning a set of identifiers) and the SQL text search function 'contains' (one call per given identifier, result is true/false). This can be overcome by replacing or supplementing the 'contains' function with a different type of user-defined function that matches the characteristics of the text engine access. Based on the concepts and syntax introduced in SQL3, we can define a new user-defined function that takes as its arguments a text search pattern and information about the scope of the search, and produces as a result a table of identifiers for the documents matching the search expression. The following syntax could be used to create such a function in SQL, which is named 'containstable'.[2]

```
CREATE FUNCTION containstable
    (schema     VARCHAR(8),
     table      VARCHAR(18),
     column     VARCHAR(18),
     searcharg  LONG VARCHAR)
RETURNS TABLE(resultid  db2texth);
```

The schema[3], table, and column names are parameters supplied by the user to specify the scope of the text search. Internally, this information can be used by the 'containstable' function to determine the information (such as the external text index name) that needs to be supplied to the text search engine for specifying the context of the search.

1. Additionally, DB2 supports a 'fenced' execution mode for 'untrusted' UDFs, where the UDF runs in its own process, separate from the DB engine.

2. The concept of table functions is available in IBM's DB2 Universal Database product

3. A schema in DB2 is simply a collection of named objects. The same table name can be used multiple times in different schemas, denoting different tables.

This information could be stored either in an additional database table serving as a system catalog for text indices, or in an external file. The result of the function call is a table, whose rows identify the matching documents.
A query using this new UDF when written in standard SQL3 would look like the following:

```
SELECT proj_no, title
FROM compschema.projects
WHERE description_id IN
    (SELECT resultid
     FROM TABLE(containstable('COMPSCHEMA',
            'PROJECTS',
            'DESCRIPTION_ID',
            ' "databasc" IN SAME SENTENCE AS
              "object-relational" '))
        AS restab (resultid))
```

This query produces the same results as the original query introduced in Section 2 using 'contains'. The 'containstable' function producing a table of ids can be used in the FROM clause of the SELECT statement, and referenced just like any other table, as can be seen in the subquery in the above query.
An alternative usage of this table function is the following query, which uses a more efficient join instead of a subquery. A good query optimizer would also be capable of internally producing this form of the query out of the previous one.

```
SELECT proj_no, title
FROM compschema.projects,
    TABLE(containstable('COMPSCHEMA',
            'PROJECTS',
            'DESCRIPTION_ID',
            ' "database" IN SAME SENTENCE AS
              "object-relational" '))
        AS restab (resultid)
WHERE description_id = resultid
```

As can be seen from the queries, in order to retrieve information about the matching projects, a join with the projects table using the id columns has to be performed. For performance reasons, an index on the identifier column should be available, so that a more efficient join method can be chosen by the optimizer.
The significant advantage of using the table function 'containstable' is the performance gain. Since the resulting identifiers can be directly picked up by the database engine to locate the tuples in the projects table, the table scan that had to occur in the original 'contains' solution is avoided. In other words, the index of the text search engine is really utilized by the database engine to directly determine result tuples in the projects table. Note that a database index has to exist on the identifier column (description_id) so that the resulting indentifiers can be used for fast lookup.
But still there are disadvantages in this approach:

- The user has to decide to use the 'containstable' function and the different form of query instead of the original one using 'contains' as introduced in Section 2. Moreover, existing query front-end tools that generate SQL queries currently cannot create the above syntax involving table functions. It would therefore be nicer if the DBMS (i.e., the optimizer) could automatically apply a rewrite, if the underlying base table is large enough.
- The containstable UDF works very well for base tables. However, in the case that we have a view that, for instance, produces a union of two text tables, the usage of the 'containstable' function may cause problems. This is because the view itself is not associated with a text index at all, but the two text tables can be associated with different text indices. If the user supplies the schema, table and column name of the view, the containstable function would need to break down the view definition to the columns of the base tables involved. This can be a very tedious tasks involving the lookup of view definitions in system catalogs and semantic analysis of the view definitions.

Therefore, we propose an automatic rewrite approach involving the DB optimizer.

4 The Automatic Rewrite Approach

The above described containstable UDF exhibits exactly the 'properties' that one usually finds with standard DB indexing. For the evaluation of a certain predicate (or function returning a boolean value), a special function (index lookup) can be applied that yields an identification of the matching values. For locating the applicable index, one needs to know the schema, table, and column for the item involved in the predicate, plus a value for the actual lookup. These are exactly the input parameters for 'containstable'.
If the optimizer knows about the existence of this function and about the fact that it can be exploited when evaluating the 'contains' function, then an automatic rewrite can be performed by the optimizer. This can happen after view expansion, which solves the second problem described above.
Note that the user still has to supply the definition of the table function, such as 'containstable'. The table function can hardly be generated automatically, because it follows a different processing model than a scalar function.

4.1 Defining Index Functions

Assume the following definition for the UDF 'contains':

```
CREATE FUNCTION
    contains(text db2texth, searcharg VARCHAR)
RETURNS BOOLEAN
```

Then this information required for performing the rewrite can be communicated to the DBMS through the following extensions in the definition of the 'containstable' UDF:

```
CREATE FUNCTION containstable(
    schema      VARCHAR(8),
    table       VARCHAR(18),
    column      VARCHAR(18),
```

```
    searcharg     LONG VARCHAR)
RETURNS TABLE(resultid   db2texth)
INDEX FUNCTION FOR
    contains (id db2texth, arg VARCHAR)
INDEX CHECK index_exists (
    VARCHAR(8), VARCHAR(18), VARCHAR(18))
COLUMNS id AS resultid
VALUE arg
```

Some comments on the above extensions:

- INDEX FUNCTION FOR contains (id db2texth, arg VARCHAR)
  This specifies that the 'containstable' function is a rewrite alternative ('index function') for the UDF 'contains' with the given parameter types.
- INDEX CHECK index_exists (VARCHAR(8), VARCHAR(18), VARCHAR(18))
  For the case that the creation of the external text index is not communicated to the DBMS (e.g., using an extension of the 'CREATE INDEX' statement), this clause specifies a boolean UDF that can be used to determine whether a user-defined index has actually been defined on a certain column. The function parameters are the schema name, table name, and column name.
  Such a function can be used **at compile time** to determine whether an index exists (result value = true) and therefore the rewrite can be performed, or whether this is not the case (result = false).
- COLUMNS id AS resultid
  This specifies (1) a parameter position of the 'original' UDF (contains), and (2) a field name of the table returned by the 'index UDF'. The parameter in (1) should hold the column item in the original query. In other words, this is the name of the parameter that is replaced by the name of the indexed column in the UDF call. The field name in (2) specifies the name of the corresponding column/field in the result table produced by the 'index UDF'.
  It is a default assumption, that the index UDF has three input parameters for the given column, which take the schemaname, tablename, and columnname. One may think of index UDFs that involve more than one table column. In this case, more than one parameters can be specified and the number of 'default' parameters for the index UDF would be 3 * # of columns.
- VALUE arg
  This specifies the parameter position of the 'original' UDF (contains), which will hold the 'value' we are using for the index lookup.

Given our sample query

```
SELECT proj_no, title
FROM projects
WHERE contains(description_id,
    ' "database" IN SAME SENTENCE AS
     "object-relational" ')
```

and the above definition of the function 'containstable' as an index function, the rewrite process would be performed in the following steps.

1. Based on the function signature (i.e., name and parameter types) of the 'contains' function, the 'containstable' function can be determined as a possible index function.
2. In the query, the 'description_id' column is used as the first parameter of the 'contains' function. This matches the position of the formal parameter 'id' for the contains function, specified in the clause 'COLUMNS id AS resultid' of the 'containstable' function.
3. In the query, the second argument of the 'contains' function call is a text literal (constant). This matches the position of the formal parameter 'arg', specified in the clause 'VALUE arg' of the 'containstable' function.
4. Using the schema, table, and column names of the 'description_id' column as parameters, the check function 'index_exists' is called. The following rewrite steps are only performed if the function evaluates to true.
5. Based on the correspondence of the formal parameter names specified in the index function definition, the 'contains' function call can be replaced by an IN predicate. The first operand of the IN predicate is the 'description_id' column, because its formal parameter 'id' in the 'contains' function is associated with the result column name of the index function (resultid) in the clause 'COLUMNS id AS resultid'. The second operand of the IN predicate is the subquery statement involving the call of the index function 'containstable' as a table function. The names of the schema, table, and column involved in the original function call, together with the search argument value are passed as arguments of the 'containstable' function.
6. This results in the following rewritten query:

```
SELECT proj_no, title
FROM compschema.projects
WHERE description_id IN
    (SELECT resultid
     FROM TABLE(containstable('COMPSCHEMA',
              'PROJECTS',
              'DESCRIPTION_ID',
               ' "database" IN SAME SENTENCE AS
                "object-relational" '))
         AS restab (resultid))
```

4.2 Rewrite Process

Based on the syntax given above for defining the index function, we can now define in a general form, how a query can be rewritten to replace the appearance of a (scalar) search function by an equivalent search condition involving the index function.

Consider the following 'template' for an index function definition

```
CREATE FUNCTION <index-function-name>
        (<schema>  VARCHAR(8),
         <table>   VARCHAR(18),
         <column>  VARCHAR(18),
```

```
        <value>    <valuetype>)
RETURNS TABLE (<resultcol>  <columntype>)
INDEX FUNCTION FOR <original-function>
        (<param1>  <columntype>,
         <param2>  <valuetype>)
INDEX CHECK <index-check-fct>
        (<schema>  VARCHAR(8),
         <table>   VARCHAR(18),
         <column>  VARCHAR(18))
COLUMNS <param1> AS <resultcol>
VALUE <param2>
```

Let's assume that we have an occurrence of the original UDF with the following pattern.

SELECT ... FROM ... WHERE <original-function> (<colname>, <searcharg>)

where the formal parameter name for <colname> in <original-function> is <param1>, and the formal parameter name for <searcharg> is <param2>. Assume that <schemaname> and <tablename> are the schema and table names for the column <colname>, and that <restablename> is an arbitrary (temporary) table name generated by the optimizer for the rewrite.
Then we can rewrite the query in the following way:

```
if (<index-check-fct>(<schemaname>, <tablename>,
                      <colname>) = 'true')
replace '<original-function>(<colname>,<searcharg>)'
with '<colname> IN
    (SELECT <restablename>.<resultcol>
     FROM TABLE(<index-function-name>(
                    <schemaname>,
                    <tablename>,
                    <colname>,
                    <searcharg>)
                AS <restablename>)'
```

The rewrite replaces the occurrence of the original function with an IN predicate involving a subquery over the table produced as the result of the index function. As already pointed out earlier, an equivalent form of the query produced by the rewrite would avoid the subquery, but position the call of the index function in the FROM clause of the query and use a join predicate instead of the IN predicate. For our proposal, we assume that the transformation of the subquery to the join can be left to the optimizer as a standard rewrite optimization, and therefore does not need to be incorporated explicitly into the index function rewrite.

4.3 Queries Involving Views

As one can see from the above description of the rewrite process, this rewrite is 'local' in the sense that it simply replaces the occurrence of one predicate with another one that involves a subquery. This can be seen as a local 'predicate expansion', which can be applied by a reasonably capable optimizer much in the same way as a view expansion. However, there is one constraint in terms of when this simple rewrite can be applied: In order to handle queries over views correctly, the rewrite can only be applied after the optimizer has expanded the view definitions and merged them with the original predicates of the query. Consider the following simple view definition:

```
CREATE VIEW compschema.expensive_projects AS
SELECT *
FROM compschema.projects
WHERE budget > 500 000
```

and the following query

```
SELECT proj_no, title
FROM compschema.expensive_projects
WHERE contains(description,
    ' "database" IN SAME SENTENCE AS
     "object-relational" ')
```

Applying the proposed index function rewrite before the view expansion would result in the following rewritten query:

```
SELECT proj_no, title
FROM compschema.expensive_projects
WHERE description_id IN
    (SELECT resultid
     FROM TABLE(containstable('COMPSCHEMA',
                'EXPENSIVE_PROJECTS',
                'DESCRIPTION',
                ' "database" IN SAME SENTENCE AS
                  "object-relational" '))
          AS restab (resultid))
```

However, the containstable function will fail to return the desired results in this case, because the table name supplied as one of its parameters ('EXPENSIVE_PROJECTS') is not the name of a base table, but the name of the view. Therefore, a lookup to determine the name of the index associated with the queried table will fail, since only base tables can be indexed. Consequently, the index function rewrite can only be applied after the optimizer has brought the query into the following form (or into an internal representation equivalent to this query) by expanding the view definition and merging the WHERE clauses.

```
SELECT proj_no, title
FROM compschema.projects
WHERE     budget > 500 000
    AND   contains(description,
                ' "database" IN SAME SENTENCE AS
                  "object-relational" ')
```

Then, the application of the index function rewrite yields the following, correct form.

```
SELECT proj_no, title
FROM compschema.projects
WHERE     budget > 500 000
    AND   description_id IN
    (SELECT resultid
     FROM TABLE(containstable('COMPSCHEMA',
                'PROJECTS',
```

```
        'DESCRIPTION',
        ' "database" IN SAME SENTENCE AS
          "object-relational" '))
    AS restab (resultid))
```

5 Further Discussion

Optimizer Considerations

The rewrite mechanism for index function integrates very well with existing optimizer technology, mainly because of two reasons:

- The required transformation affects only the scope of a single predicate in a query, and replaces the predicate entirely with another one. No complex transformations in the scope of the entire query are required. Therefore, the implementation of the rewrite in the scope of a relational optimizer is inexpensive, and does not come in the way with existing rewrite optimization rules.
- The rewrite utilizes existing language constructs (i.e., user-defined table functions), which, if supported by the database engine, are already known to the optimizer in terms of execution costs, statistics, etc. For example, DB2 Universal Database [Cha96, Dav96] allows the user to give information to the optimizer about the execution costs of external, user-defined functions. Since index functions are nothing more than standard user-defined table functions, the optimizer can use cost information supplied for them in the further process of optimizing the query. In other words, no additional extensions to the optimizer are required for communicating information about costs and statistics.

Generalization for arbitrary predicates

The above approach supports the rewrite of user-defined functions that return a boolean value. What if a function returns other (numeric or non-numeric) values, and appears as an operand of an arbitrary SQL predicate? Assume the following definition for the UDF 'rank':

```
CREATE FUNCTION rank (id  db2texth,
                      arg  LONG VARCHAR)
RETURNS DOUBLE PRECISION
```

This function behaves like the contains function, but returns a rank value instead of a boolean value, describing how well a document meets the text search criteria. For example, the query

```
SELECT proj_no, title
FROM projects
WHERE rank (description,
            ' "database" IN SAME SENTENCE
AS "object-relational" ')
          > 0.5
```

would retrieve all text information for documents that match the given search argument with a rank value > 0.5 (all rank values range between 0 and 1).

Our approach can be generalized to allow index functions that can be applied in this case as well. For this type of support, the index function would need to take additional arguments for capturing the predicate and the additional operands in the predicate. The CREATE FUNCTION statement for the indexing function would then look like the following one:

```
CREATE FUNCTION ranktable(
    schema      VARCHAR(8),
    table       VARCHAR(18),
    column      VARCHAR(18),
    searcharg   LONG VARCHAR,
    predicate   VARCHAR(18),
    rankval     DOUBLE PRECISION)
RETURNS TABLE(resultid   db2texth)
INDEX FUNCTION FOR
    rank(id db2texth, arg LONG VARCHAR)
INDEX CHECK index_exists
    (VARCHAR(8), VARCHAR(18), VARCHAR(18),
     VARCHAR(18))
COLUMNS id AS resultid
VALUE arg
```

There are two significant changes (marked by putting them in **bold** face) when compared with the approach outlined above for the boolean functions:

- Additional parameters have been added for communicating the predicate (in textual form) and the 2nd operand involved in the predicate to the indexing function. At run-time, these parameters will hold the predicate (e.g., '>') and the second parameter (e.g., 0.5) of the comparison in which the original function call appears.
- The function for checking at compile-time, if an index exists, has been extended by an additional parameter that holds the predicate involved in the comparison (e.g., '>'). This is used by the 'index_exists' function to determine, whether the user-defined index functionality can handle the predicate at all. If the predicate cannot be interpreted by the indexing function, then the 'index_exists' function can return an appropriate error code at compile-time to prohibit the rewrite.

The automatic rewrite for the above query would then result in the following query:

```
SELECT proj_no, title
FROM compschema.projects
WHERE description_id IN
    (SELECT resultid
     FROM TABLE(ranktable('COMPSCHEMA',
                'PROJECTS',
                'DESCRIPTION',
                ' "database" IN SAME SENTENCE AS
                  "object-relational" ',
                '>',
                0.5))
          AS restab (resultid))
```

In other words, the predicate '>' and the second operand '0.5' would be passed to the index function as additional

parameters. The lower bound of 0.5 for the rank value can then either be passed to the external search engine (provided that the programming API of the text search engine supports this), or it can be applied on the results delivered by the text search engine in the table function itself, which would return only those documents whose rank value exceed the given threshold.

6 Related Work

The presented approach allows the efficient exploitation of content-specific indexing and search capabilities of existing, stand-alone search engines in databases, without having to add new, content-specific access methods to the database or map content-specific indices onto the existing database access methods.

[CS93] addresses the same problem, and presents a solution based on user-defined, logical rewrite rules for specifying logical equivalences of subqueries. In contrast, our approach focuses on the easy and straightforward specification of a certain type of equivalences that can also be handled more efficiently by the query engine in the query rewrite phase, without introducing a logic-based rewrite rule notation. Moreover, we introduce capabilities to supply the table functions in the rewritten part of the query with information available to the query processing engine, such as table name and column name of the columns involved in the query, and with the capability to register index check functions (such as 'index_exists') that allow the query engine to check on prerequisites to be met before the rewrite takes place.

The approaches presented in [DDSS95] and [GFHR96] for integrating text search and databases have been developed on the basis mapping the text index itself into relational tables. In other words, an (enhanced) inverted word list of text documents is stored in a relational table, which can be indexed using the standard indexing techniques of the DBMS. [DDSS95] uses an enhancement of the database engine for this purpose, which is essentially a capability to store a table as 'index only' or as an inverted table structure. In both approaches, the user or application assumes responsibility for updating index information, if the original text documents change. Moreover, the application has to be aware of the index model and the correlation of text tables and index tables in formulating text search queries. Essentially, this approach completely exposes the structure of the content-specific index and requires to reimplement content-specific search within the database system. Using this mapping scheme, the support that can be provided by the database system is limited to a text search involving a single keyword. Complex search patterns need to be mapped to a combination of independent relational operations on the underlying base tables, making additional optimizations usually performed by search engines not applicable.

The approach described in [LS88], also in the realm of textual databases, which is based on extensions of indexing support in the POSTGRES system [Sto86], essentially suffers the same drawbacks. An extended indexing support is provided that permits a user-defined function to produce a list of values for given column value, which are then stored in the index entries instead of the original column value. This is used to produce a list of keywords found in a text document, resulting in an index structure similar to an inverted word list. The same limitation in terms of the complexity of search and the required reimplementation of additional search capabilities within of the DBMS as described above apply here as well.

A number of database access methods have been developed for content-specific search, such as the R-tree [Gut84] and the P-tree [Jag90] for spatial data, or D-trees [Dav96b] for textual data. In the (rare) case that a database system supports such access methods, they are 'hard-wired' into the database engine, leaving no room for adjustments to the specific requirement and advantages of the indexing technology 'owned' by a search engine vendor.

Adding new access methods to a database engine is a very complicated and expensive task. [HNP95] and [KMH97] present an access method 'template' called generalized search trees, which can easily be 'instantiated' by plugging in various 'operations' into the generic template. While this approach seems to be very promising, it is proposing a framework for a tight integration of user-defined access methods into the DB system, whereas our approach concentrates on making existing access structures of external search engines efficiently exploitable in SQL queries.

[HS93] addresses the optimization of queries involving expensive predicates, such as external function invocations. By exploiting cost information about the predicate invocations, the query processor will evaluate such expensive predicates as late in the plan as possible. Such techniques rely only on the availability of cost information for the execution of UDFs, and do not attempt to replace a query predicate by another predicate or subquery. They therefore complement our approach in that they can be used to further optimize both the original query plan, as well as the one resulting from our proposed rewrite, and chose the best one in the end.

7 Summary

Object-relational database systems have started to leave the research labs and become a reality in the marketplace. These systems are capable of language extension that permit suppliers of content-specific search technology, such as fulltext retrieval engines, to 'plug' into the database engine in order to extend the content management and search capabilities of SQL.

In this paper, we have presented an approach that supports this type of plug-in extensibility in an easy and efficient manner. Based on the concept of user-defined table functions, which can be registered in the database engine as so-called index functions, a search technology provider can integrate search as well as indexing support into the database engine without significant impact to the original, content-specific search engine, and without sacrificing integration at the query language (SQL) and query execu-