Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8*i*

Jagannathan Srinivasan , Ravi Murthy, Seema Sundara, Nipun Agarwal, Samuel DeFazio
Oracle Corporation
{jsriniva, rmurthy, spsundar, nagarwal, sdefazio}@us.oracle.com

Abstract

Extensible Indexing is a SQL-based framework that allows users to define domain-specific indexing schemes, and integrate them into the Oracle8i server. Users register a new indexing scheme, the set of related operators, and additional properties through SQL data definition language extensions. The implementation for an indexing scheme is provided as a set of Oracle Data Cartridge Interface (ODCIIndex) routines for index-definition, index-maintenance, and index-scan operations. An index created using the new indexing scheme, referred to as domain index, behaves and performs analogous to those built natively by the database system. Oracle8i server implicitly invokes user-supplied index implementation code when domain index operations are performed, and executes user-supplied index scan routines for efficient evaluation of domain-specific operators. This paper provides an overview of the framework and describes the steps needed to implement an indexing scheme. The paper also presents a case study of Oracle Cartridges (InterMedia Text, Spatial, and Visual Information Retrieval), and Daylight (Chemical compound searching) Cartridge, which have implemented new indexing schemes using this framework and discusses the benefits and limitations.

1. Introduction

Typical database management systems support a few types of access methods (E.g. B^+-Trees [Com79], Hashed Index [HK95]) on some set of pre-defined data types (numbers, strings, etc.). In recent years, databases are being used to store different types of data like text, spatial, image, video and audio. For these complex domains, there is a need for

- indexing complex data types for efficient evaluation of existing relational operators such as (=,<,>)
- supporting domain-specific operations and specialized indexing techniques. For instance, in the spatial domain, efficient processing of the *Overlaps* operator requires a specialized indexing structure such as R-trees [Gut84].

However, building a database server with native support for all possible kinds of complex data and indexing is not viable. The solution is to build an extensible server which allows users to define new index types. In Oracle8*i*, this is achieved through *Extensible Indexing*, a SQL-based interface that allows users to define domain-specific operators and indexing schemes, and integrate them into the Oracle8 *i* server.

Oracle provides a set of pre-defined operators which include arithmetic operators (+, -, *, /), comparison operators (=, >, <) and logical operators (NOT, AND, OR). These operators take as input one or more arguments (or operands), and return a result. User-defined operators, identified by names (e.g. Contains), are similar to built-in operators, except that their implementation is provided by the user. After a user has defined a new operator, it can be used in SQL statements like any other built-in operator. For example, if the user defines a new operator *Contains*, which takes as input a text document and a keyword search expression, and returns TRUE if the document satisfies the search expression, we can write a SQL query as :

```
SELECT * FROM Employees WHERE
 Contains(resume, 'Oracle AND UNIX')[1];
```

Oracle uses indexes to efficiently evaluate some built-in operators. For example, a B-tree index can be used to evaluate the comparison operators =, > and <. In Oracle8*i*, through the extensible indexing framework, user-defined domain indexes can be used to efficiently evaluate user-defined operators. The framework to develop new index types is based on the concept of cooperative indexing[D+95], where an application and the Oracle server cooperate to build and maintain indexes. The application software is responsible for defining the index structure, maintaining the index content during load and update operations, and searching

[1] Currently, Oracle8*i* SQL syntax requires specifying Contains(…) = 1. However, for better readability, all the examples in the paper use predicates of the form Contains(…).

the index during query processing. The index structure itself can either be stored in Oracle database as tables, or externally in files.

The extensible indexing framework consists of the following components:

- *Operators*: Queries and data manipulation statements can involve user-defined operators, like the *Overlaps* operator in the spatial domain. In general, user-defined operators can be bound to functions. However, operators can also be evaluated using indexes. For instance, the equality operator can be evaluated using a hash index. Similarly, user-defined operators can be evaluated using domain-index based scan.
- *Indextype*: A new schema object, called an Indextype, specifies the routines that manage all the aspects of application-specific index, namely, index definition, index maintenance, and index scan operations. This schema object enables the Oracle server to establish a user-defined index on a column of table or attribute of an Object. It also specifies the set of user-defined operators that can be evaluated using a domain index defined using this indextype.
- *Domain Index*: Using the Indextype schema object, an application-specific index can be created. Such an index is called a domain index since it is used for indexing data in application-specific domains. A domain index is created, managed, and accessed by routines supplied by an indextype. This is in contrast to B-tree indexes maintained natively by Oracle.
- *Index-Organized Tables*: This feature enables users to define, build, maintain, and access indexes for complex objects using a table metaphor. To the user, an index is modeled as a table, where each row is an index entry. For detailed information on index-organized tables, see [Ora99].

To illustrate the role of each of these components, let us consider a text domain application. Suppose a new indextype `TextIndexType` has been defined which contains routines for managing and accessing the text index. The text index is an inverted index that stores the occurrence list for each token in each of the text documents. The text cartridge[2] also defines the `Contains` operator for performing content-based search on textual data. The `TextIndexType` provides scan routines to evaluate the `Contains` operator using a domain index. A function that implements the `Contains` operator (i.e. without using the index) is also required. A domain index can be created on `resume` column, containing textual data, of `Employee` table as follows:

```
CREATE TABLE Employees(name VARCHAR(128),
 id INTEGER, resume VARCHAR2(1024));
CREATE INDEX ResumeTextIndex ON
 Employees(resume)
 INDEXTYPE IS TextIndexType;
```

Oracle server invokes the routine corresponding to the create index method in the `TextIndexType`, which results in the creation of a table to store the occurrence list of all tokens in the resumes (essentially, the inverted index data). Whenever an `Employees` row is inserted, updated, or deleted, the inverted index, modeled by `ResumeTextIndex`, is automatically maintained by invoking routines defined in `TextIndexType`. Content-based search on the `resume` column can be performed as follows:

```
SELECT * FROM Employees WHERE
 Contains(resume, 'Oracle AND UNIX');
```

The index-based implementation of the `Contains` operator can take advantage of the previously built inverted index. Specifically, Oracle server can invoke routines specified in `TextIndexType` to search the domain index for identifying candidate rows, and then do further processing, namely, filtering, selection, and fetching of rows. Note that , if the Oracle server chooses to not use the index defined on `resume` column, the above query will be evaluated using the non-indexed implementation of the `Contains` operator. In such a case, the filtering of rows will be done by applying the functional implementation on each `resume` instance of the table.

In summary, the extensible indexing interface allows encapsulating user provided index management routines as an indextype schema object, supports defining a domain (application-specific) index on table columns, and provides index based processing of application-specific operators. This interface enables a domain index to operate essentially the same as any other Oracle built-in index. The primary difference is that Oracle server will invoke application code specified as part of indextype to create, drop, truncate, modify, and search a domain index. It should be noted that an index designer may choose to store the index in files, rather than in database tables. The SQL interface for extensible indexing makes no restrictions on the location of the index data, only that the application adhere to the protocol for index definition, maintenance and search operations.

The rest of the paper is organized as follows. Section 2 presents the extensible indexing framework. Section 3 discusses typical usage of this framework and presents a case study of cartridges that have implemented new indexing schemes using this framework. Section 4

[2] Data Cartridges refer to server-managed components that provide integrated support for a specific domain. Typically they include user-defined types, functions, operators, &indextypes.

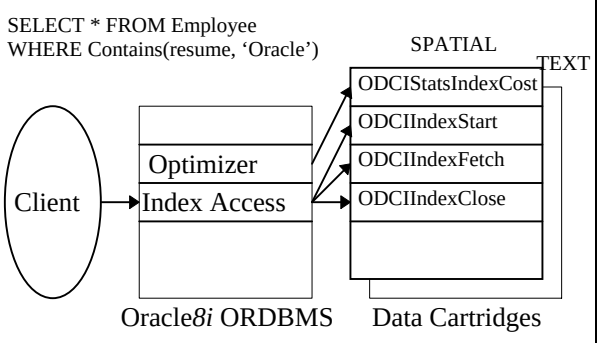

Figure 1 : Oracle8*i* Extensibility Architecture

covers the related work, section 5 identifies limitations, and section 6 summarizes the paper.

2. Extensible indexing framework

This section gives an overview of the extensible indexing framework, describes the components, and discusses performance issues and distinguishing aspects of our approach.

2.1. Overview

Figure 1 shows the overall architecture of the extensibility framework in Oracle8*i*. When the Oracle server receives a SQL request from a client, the server calls the appropriate user-defined routines that have been registered by a domain with the server. For example, if the SQL query contains a user defined operator that has been registered by the text domain, the indexing component of the Oracle server will call the index scan routines (ODCIIndexStart/Fetch/Close), that have been implemented and registered by the text domain. Similarly the optimizer component of Oracle8i will call the cost (ODCIStatsIndexCost) and selectivity (ODCIStatsSelectivity) routines[ODC99].

We envision two classes of users namely,

- *Cartridge Developer*: Defines a new indexing scheme and operators, and registers them with the database. The steps which the indextype designer is expected to undertake include providing functional implementations for user-defined operators, creating the operators, defining a type that implements the ODCIIndex routines for index definition, maintenance and scan operations, and finally creating the indextype schema object.
- *End User*: Uses the indextype created by the indextype designer to build domain indexes and operate on the domain data.

The following sections describe in detail, the steps that each of the above mentioned class of user needs to take.

2.2. Steps performed by cartridge developer

2.2.1. Functional implementation of operators. Define and code functions to support operators which eventually would be supported by the indextype. If the optimizer does not choose the domain index scan to evaluate the user-defined operator, the evaluation of the operator transforms to execution of this function. Suppose our text indexing scheme supports an operator *Contains*, that takes as parameters a text value and a key, and returns a boolean value indicating whether the text contains the key.

```
CREATE FUNCTION TextContains(
 Text IN VARCHAR2, Key IN VARCHAR2)
RETURN BOOLEAN AS
BEGIN
 ...
END TextContains;
```

2.2.2. Operator definition. A user-defined operator is a top level schema object in Oracle8*i*, and has a set of one or more bindings associated with it. An operator binding identifies the operator with a unique signature (via argument data types), and allows associating a function that provides an implementation for the operator. Each of the bindings can be evaluated using a user-defined function which could be a stand-alone function, package function or a member method of an object type.

For example, the operator `Contains` can be created in the `Ordsys` schema, with its bindings and the corresponding functional implementations.

```
CREATE OPERATOR Ordsys.Contains
 BINDING (VARCHAR2, VARCHAR2)
RETURN NUMBER USING TextContains;
```

A user defined operator can be *supported* by one or more user-defined indextypes. This means that a domain index of an indextype can be used in efficiently evaluating the operators it supports.

2.2.3. ODCIIndex definition & implementation. Define a type or package that implements the index interface, ODCIIndex. These methods handle the definition, maintenance and scan of the domain indexes. The domain index metadata information such as the index name, table name, and names of the indexed columns and their data types, are passed in as arguments to all the ODCIIndex routines. The indextype designer is expected to implement all the ODCIIndex routines as methods within a user-defined type.

```
CREATE TYPE TextIndexMethods AS OBJECT (
 STATIC FUNCTION ODCIIndexCreate(...)
 ... );
```

```
CREATE TYPE BODY TextIndexMethods AS (
 ... );
```

ODCIIndex definition methods. Index Definition methods allow specification of create, alter, truncate, and drop behaviors through ODCIIndexCreate /Alter/Truncate/Drop routines. Typical actions performed in these functions are to create tables/files to store index data, alter tables storing index, truncate tables storing index data and drop the tables storing index data respectively. For example, in ODCIIndexCreate, the cartridge developer can create a table to store the index data, and if the base table containing the data to be indexed is non-empty, then do a series of inserts to insert index data into the index table.

ODCIIndex maintenance methods. Index Maintenance methods allow specification of index insert, update, and delete behaviors through ODCIIndexInsert /Update/Delete routines. These routines are passed in the new and/or old value for the indexed column and the row identifier of the corresponding row. The typical actions performed are to generate the index entries from the column values, and perform the corresponding insert /update/delete on the table storing index data. For example, ODCIIndexUpdate should delete the entries corresponding to the old indexed column value in the table storing index data, and insert the new entries corresponding to the new indexed column value. Essentially, these methods maintain the tables that store index data to be in sync with the base table.

ODCIIndex scan methods. Index Scan methods allow specification of an index-based implementation for evaluating predicates containing supported operators. An index scan is specified through three routines, ODCIIndexStart/Fetch/Close, which can perform initialization and start the index scan, fetch rows satisfying the predicate, and clean-up once all rows satisfying the predicate are returned, respectively. A typical action performed when ODCIIndexStart is invoked is to parse and execute SQL statements that query the tables storing the index data. It could also generate some set of result rows to be returned later when the fetch is invoked. ODCIIndexFetch can look at the set of result rows computed during start phase and return the "next" row identifier of the row that satisfies the operator predicate. The fetch method supports returning a single row or a batch of rows in each call. The end of the scan can be indicated by returning a null row identifier. ODCIIndexClose performs any clean-ups etc.

The index scan routines share a context that is modeled as an instance of an object type that is passed in and out of every scan routine. The scan context returned by ODCIIndexStart is passed back to subsequent invocations of ODCIIndexFetch and ODCIIndexClose. To accommodate varying sizes of the scan context without sacrificing performance, Oracle8*i* supports two mechanisms for scan context:

1. *Return State* : If the state to be maintained is small, it can be returned back to Oracle server as the output object argument.
2. *Return Handle* : If the state to be maintained is large, (e.g. a subset of the results), a temporary workspace (primarily memory resident, but can be paged to disk) can be allocated for the duration of the statement to save the state. In this case, a handle to the workspace can be returned back to Oracle server, instead of the entire scan state.

There can be two general kinds of scan routine implementations.

1. *Precompute All* : Compute the entire result set in ODCIIndexStart. Iterate over the results returning a row at a time in ODCIIndexFetch. This is generally the case for operators involving some sort of ranking over the entire collection, etc. Evaluating such operators would require looking at the entire result set to compute the ranking, relevance, etc. for each candidate row.
2. *Incremental Computation* : Compute the result rows a set at a time as part of ODCIIndexFetch. This is generally the case with operators which can determine the candidate rows one at a time without having to look at the entire result set.

Multiple sets of invocations of operators can be interleaved. At any given time, a number of operators can be evaluated using the same indextype routines.

2.2.4. Indextype definition. The purpose of an indextype is to enable efficient search and retrieval functions for complex domains, such as text, spatial, and image. An indextype is analogous to the B-tree or bit-mapped indexing schemes that are natively implemented by the Oracle server. The essential difference is that the implementation for an Indextype is provided by user software, as opposed to the Oracle server kernel routines. The cartridge developer implements the ODCIIndex routines consisting of a set of index definition, maintenance and scan routines.

The CREATE INDEXTYPE statement is introduced for registering new indexing schemes. Once the type that implements the ODCIIndex routines has been defined, a new indextype can be created by specifying the list of operators supported by the indextype, and referring to the type that implements the ODCIIndex routines. E.g.

```
CREATE INDEXTYPE TextIndexType
 FOR Contains(VARCHAR2, VARCHAR2)
```

```
USING TextIndexMethods;
```

defines the new indextype `TextIndexType` which supports the `Contains` operator, and whose implementation is provided by the type `TextIndexMethods`.

2.3. Steps performed by the end user

Once an indextype has been created, the end user can create a domain index on a column of a table just like a B-tree index. E.g.

```
CREATE INDEX ResumeTextIndex ON
 Employees(resume)
 INDEXTYPE IS TextIndexType
 PARAMETERS (':Language English :Ignore the
              a an');
```

The INDEXTYPE clause specifies the indextype to be used. The PARAMETERS clause identifies any parameters for the domain index, specified as a string. In the above example, the parameters string identifies the language of the text document (thus identifying the lexical analyzer to use), and the list of stop words which are to be ignored while creating the text index. Similarly, a domain index can be altered using ALTER INDEX statement. For example:

```
ALTER INDEX ResumeTextIndex
 PARAMETERS (':Ignore COBOL');
```

Once a domain index is created, there is no change to the SQL the end user needs to issue for other index data definition and manipulation operations

The end user can use the user-defined operators anywhere built-in operators can be used. For example, user-defined operators can be used in the following : select list of a SELECT command, the condition of a WHERE clause, the ORDER BY and GROUP BY clauses. A user-defined operator can also be a join condition.

2.4. Domain index processing

This section describes the processing in the Oracle8*i* server to handle the domain index definition, maintenance and scan.

2.4.1. Domain index definition and maintenance. When a domain index is created, the Oracle8*i* server creates the data dictionary entries pertaining to the domain index and invokes the ODCIIndexCreate() method, passing it the uninterpreted parameter string. Similarly, when a domain index is altered, ODCIIndexAlter() method is invoked. There is no explicit statement for truncating a domain index. However, when the corresponding table is truncated, the truncate method specified as part of the indextype is invoked. And when an instance of a domain index is dropped, the ODCIIndexDrop method is invoked.

When the base table is updated, all domain indexes built on columns of the table are implicitly maintained. For example, when existing rows are updated, the ODCIIndexUpdate() methods are invoked with the old and new values of the indexed columns and the corresponding row identifiers. Similarly, insert and delete operations result in ODCIIndexInsert/Delete methods being invoked with the new/old values of the indexed column and the row identifier.

2.4.2. Query optimization. When an operator is invoked, the evaluation of the operator is, by default, transformed to the execution of one of the functions bound to it. However, the operators appearing in the WHERE clause can also be evaluated by performing an index scan, as opposed to a table scan, using the scan methods provided in the indextype implementation. An indextype supports efficient evaluation of those operator predicates, which can be represented by a range of lower and upper bounds on the operator return values. Specifically, predicates of the form

op(...) relop <value expression>,
where relop in {<, <=, =, >=,>} and
op(...) LIKE <value_expression>

are possible candidates for index scan based evaluation. Other operator predicates, which can be converted into one of the above forms by Oracle, can also make use of the indexed evaluation. Oracle also supports the notion of *ancillary* operators to model auxiliary information that is returned from a domain index scan e.g. `score` value based on keyword matching [ODC99].

The index scan based evaluation of the operator is a possible candidate for predicate evaluation only if the operator in the predicate operates on a column that has been indexed using an indextype. For example, consider the query

```
SELECT * FROM Employees WHERE
 Contains(resume, 'Oracle');
```

The optimizer can choose to use a domain index in evaluating the `Contains` operator if, the `resume` column has an index defined on it, the index is of type `TextIndexType`, and `TextIndexType` supports the appropriate `Contains`() operator. The choice between the indexed implementation and the functional evaluation of the operator is made by the Oracle cost based optimizer using selectivity and cost functions.

Consider a slightly different query,

```
SELECT * FROM Employees WHERE
 Contains(resume, 'Oracle') AND id =100;
```

In this query, the `Employees` table could be accessed through an index on the `id` column or one on the

`resume` column. The optimizer estimates the costs of the two plans and picks the cheaper one, which could be to use the index on `id` and apply the `Contains` operator on the resulting rows. In this case, the functional implementation of `Contains()` is used and the domain index is not used. For details on how to specify cost of domain indexes, index statistics collection functions, and the selectivity of operators, refer [ODC99].

2.5. Discussion

The extensible indexing framework is primarily SQL based. The registration of a new indexing scheme, as well as its properties such as the list of supported operators, are all done through SQL data definition language extensions. The framework is based on a set of interfaces that capture the functionality of a conceptual index, and do not involve database specific concepts such as transactions, concurrency, etc. The set of routines that need to be implemented for a new indexing mechanism is available to all users of Oracle8*i***.** The indexing interface definitions are language independent, which are defined by Oracle using a form of Interface Definition Language, and SQL data types. This provides the index designer the flexibility of choosing a language (C/C++, Java, or procedural SQL - PL/SQL) that is best for the purpose.

The index designer has complete flexibility in choosing storage structures for the index. The index data can be stored within the database itself in heap tables, index-organized tables and in Large Objects(LOBs). The index data can also be stored outside the database as files, and accessed using Binary files. Based on experience, we have found that index-organized tables are commonly used as index data stores.

The index routines typically use SQL to access and manipulate index data. The SQL statements executed by the indexing logic are referred to as server *callbacks*. The callbacks exploit the performance and scalability of Oracle8*i* SQL processing. Features, such as parallelism, get used automatically without any special effort on the part of the index designer. Also, batch interfaces are provided to reduce interactions between application and server code. For example, the ODCIIndexFetch() routine can return a single or a batch of row identifiers.

When the index data is stored within the database, and is accessed and manipulated using SQL, the server functionality, in terms of concurrency control and data buffering, are also applicable to the user index data. Hence, it is not necessary for the index designer to implement low level interfaces for locking, etc. However, there are certain types of index structures that might require finer grained user control. See section 5 for a discussion on how this might be accomplished.

The transactional semantics are also automatically ensured for the user index data, if the index data resides within the database. Updates to the index data are within the same transactional boundaries as updates to the base table. However, if some of the index data is stored outside the database, we need a separate mechanism to maintain transaction semantics. Section 5 discusses one such mechanism in the form of *database events*. In order to achieve transactional behavior, there are certain restrictions on the nature of the SQL callbacks that can be made by the indextype routines. Index maintenance routines can not execute DDL statements. Also, these routines cannot update the base table on which the domain index is created. Index scan routines can only execute SQL query statements. There are no restrictions on the index definition routines.

The index maintenance and scan routines execute with the same snapshot as the top level SQL statement. This enables the index data, processed by the index method, to be consistent with the data in the base tables. Indextype routines always execute under the privileges of the owner of the index. However, for certain operations such as metadata maintenance, indextype routines may require to store information in tables owned by the indextype designer. Oracle8*i* provides a mechanism to execute certain pieces of code under the privileges of the definer, instead of the current invoker [Ora99].

In the absence of an extensible indexing like framework, many applications resort to maintaining file-based indexes for data residing in database tables. A considerable amount of code and effort is required to maintain consistency between external indexes and the related relational data, support compound queries (involving tabular values and external indexes), and to manage a system (backup, recovery, allocate storage, etc.), with multiple forms of persistent storage(files and databases). By supporting internal storage for domain indexes, the extensible indexing framework significantly reduces the level of effort needed to develop applications involving the use of complex data types.

3. Applications

This section discusses the typical usage of extensible indexing framework. and describes how various data cartridges use it to build specialized indexing mechanisms.

3.1. Typical usage

Oracle8*i* built-in indexing schemes allow creating B-tree and bitmap indexes on columns. However, these built-in indexing schemes are applicable only to scalar types. Furthermore, they only support index-based evaluation of relational operators on columns. By using the extensible indexing framework, users can augment Oracle's built-in indexing capability in several ways. Specifically new indextypes can be defined for:

- *Indexing Object Type Columns:* With built-in indexing schemes, only the individual scalar attributes can be indexed, not the entire object type column. However, object type columns can be indexed using the framework. For example, Oracle8*i* Spatial Data Cartridge[OSpa99] supports indexing spatial geometry objects. For details, see the next section.
- *Indexing Collection Type Columns (VARRAY and Nested Table Columns):* In Oracle8*i*, collection type columns cannot be indexed using built-in indexing schemes.Consider the operator Contains(VARRAY, elem_value) which returns TRUE if the VARRAY contains an element with the value elem_value. For such an operator, the user can provide both a functional implementation as well as an indextype based implementation and use it for processing queries such as:

```
SELECT * FROM Employees WHERE
 Contains(Hobbies, 'Skiing');
```

- *Indexing LOB Columns*: LOB columns store large unstructured data. They are typically used to hold multimedia data such as text, image, and video. Both top level LOB columns as well as LOB attributes embedded in an object type column can be indexed using extensible indexing. For example, the framework has been used by Oracle8*i* VIR Data Cartridge[OVIR99] to support efficient processing of *Similar* operator, which operates on columns containing image objects. An image object contains a LOB attribute to hold the image data. The Oracle8*i* VIR Data Cartridge is discussed in greater detail in the next section.
- *Indexing Scalar Columns*: The framework can also be employed to index scalar columns especially to support non-relational operators. For example, the framework has been used by Oracle8*i* *inter*Media Text[OIMT99] to index VARCHAR text columns to support efficient processing of content-based search queries. The Oracle8*i* *inter*Media Text is discussed in greater detail in the next section.

3.2. Case studies

We present some of the data cartridge implementations that use the extensibility framework and discuss how they have benefited from using the framework.

3.2.1. Oracle8*i* interMedia text cartridge. The Oracle8*i* *inter*Media Text Cartridge supports full-text indexing of text documents. The text index is an inverted index, storing the occurrence list for each token in each of the text documents. The inverted index is stored in an index-organized table, and is maintained by performing insert/update/delete on the table whenever the table on which the text index is defined is modified. The text cartridge defines an operator *Contains* that takes as input a text column and a keyword, and returns true or false depending on whether the keyword is contained in the text column or not. The benefits of extensible indexing framework can be seen by analyzing the execution of the same text query before and in Oracle8*i*. Consider an example query :

```
SELECT * FROM docs WHERE
 Contains(resume, 'Oracle');
```

In releases prior to Oracle8*i*, the text indexing code, though logically a part of the Oracle server, was not known by the query optimizer to be a valid access path. As a result, text queries were evaluated as a two step process:

1. The text predicate was evaluated first. The text index was scanned and all the rows satisfying the predicate were identified. The row identifiers of all the relevant rows were written out into a temporary result table, say *results*.
2. The original query was rewritten as a join of the original query (minus the text operator) and the temporary result table containing row identifiers for rows that satisfy the text operator, as follows :

```
SELECT d.* FROM docs d, results r
 WHERE d.rowid = r.rid;
```

In Oracle8*i*, using the extensible indexing framework, the above query is now executed in a single step and pipelined fashion. The text indexing code gets invoked at the appropriate times by the kernel. There is no need for a temporary result table because the relevant row identifiers are streamed back to the server via the ODCI interfaces. This also implies that there are no extra joins to be performed in this execution model. Further, all rows that satisfy the text predicate do not have to be identified before the first result row can be returned to the user.

The performance of text queries has improved due to : 1) Reduced I/O because of no temporary result table. 2)

Improved response time because the row satisfying the text predicate can be identified on demand. 3) Better query plans because the number of joins is reduced as there are no extra joins with temporary result tables. A decrease in the number of joins typically improves the effectiveness of the optimizer due to reduced search space. We have observed as much as 10X improvement in performance for certain search-intensive queries, after the integration using the extensible indexing framework.

3.2.2. Oracle8*i* spatial data cartridge. The Oracle8*i* Spatial cartridge allows users to store, index, and query spatial data. It supports several spatial operators that determine spatial relationships between two geometries. The spatial index consists of a collection of tiles (unit of space) corresponding to every spatial object, and is stored in an Oracle table. A spatial query to determine if two layers overlap is as follows in Oracle8*i*:

```
SELECT r.gid, p.gid  FROM roads r, parks p
 WHERE Sdo_Relate(r.geometry, p.geometry,
                  'mask=OVERLAPS');
```

When an index based evaluation is chosen for the `Sdo_Relate` operator, the operator first determines the candidate set of tiles in the parks and roads which overlap, and then applies an exact filter to these candidate rows to determine the exact set of overlapping roads and parks.

The extensible indexing framework has greatly simplified the queries and improved the usability of the Oracle spatial cartridge. Prior to Oracle8*i*, the user had to explicitly invoke PL/SQL package routines to create an index or to maintain the spatial index following a DML operation to the base spatial table. With the extensible indexing framework, the spatial index is maintained implicitly just like a built in index. Prior to Oracle8*i*, the above query had to be formulated as follows by the end user:

```
SELECT DISTINCT r.gid, p.gid FROM
 roads_sdoindex r,parks_sdoindex p
 WHERE
 (r.grpcode = p.grpcode) AND
 (r.sdo_code BETWEEN
    p.sdo_code AND p.sdo_maxcode OR
  p.sdo_code BETWEEN
    r.sdo_code AND r.sdo_maxcode) AND
 (sdo_geom.Relate(r.gid,p.gid,'OVERLAPS')
        = 'TRUE');
```

This query yields the same result as the previous query - it first determines the potential set of rows between the roads and parks layer that overlap and then the `sdo_geom.Relate` function determines exactly which of these potential rows overlap. Contrast this query with the simplicity of the query in Oracle8*i* where this logic is encapsulated in the `Sdo_Relate` operator.

The drawback of the above approach is that the querying algorithm which may be proprietary has to be exposed to the user. Furthermore, the entire logic has to be expressed as a single SQL statement, and the user is expected to know the details of the storage structures for the index. In addition to vastly simplifying the queries, the Oracle8*i* extensibility framework allows changing the underlying spatial indexing algorithms without requiring the end users to change their queries. The performance of spatial queries using the extensible indexing framework has been as good as the performance of the prior implementation.

3.2.3. Oracle8*i* VIR data cartridge. The Oracle8*i* VIR cartridge[OVIR99] supports content-based retrieval of images. Each image is represented by a signature which is an abstraction of the contents of the image in terms of its visual attributes. A set of numbers that are a coarse representation of the signature are then stored in a table representing the index data. The cartridge supports an operator *Similar* that searches for images similar to a query image. Consider an example query :

```
SELECT * FROM images T WHERE
 VIRSimilar(T.img.Signature,
            querySignature,globalcolor=0.5,
            localcolor=0.0,texture=0.5,
            structure=0.0', 10,1);
```

In releases prior to Oracle8*i*, the image cartridge had no indexing support. Hence, the operator was evaluated as a filter predicate for every row. In Oracle8*i*, using the extensible indexing framework, the `VIRSimilar` operator is evaluated in three phases- the first phase is a filter that does a range query on the index data table, the second phase is another filter that is a computation of the distance measure, and the third phase does the actual image signature comparison. Thus, the complex problem of high-dimensional indexing is broken down into several simpler components. Also, the first two passes of filtering are very selective and greatly reduce the data set on which the image signature comparisons need to be performed.

The performance of image queries has improved due to multi-level filtering process instead of doing the image signature comparison for every row, and optimization of the range query on the index data table using indexes etc. In Oracle8*i*, it is now possible to do content-based image queries on tables with millions of rows, something that was not possible in prior releases.

3.2.4. Daylight/Oracle chemistry cartridge. Daylight supports efficient indexed lookup of full molecular structure and tautomers, selection by substructure, structural similarity; and fast nearest-neighbor selection. The indexing scheme previously used a proprietary file-based index structure. The Oracle8*i* Chemistry Cartridge [DOCC99] is a re-implementation of the Daylight

chemical structure processing algorithms for use on top of Oracle RDBMS.

An extensible indexing solution was provided by storing the data within the database as LOBs. Since LOBs can be accessed and manipulated with a file-like interface[Ora99], minimal changes were required to the index management software. By using this approach, it was possible to quickly migrate file-based index data into the database, thereby providing a single data storage model for both tables and indexes.

The extensible indexing based solution scales much better that the file based indexing scheme because it minimizes intermediate write operations. Although reads against LOBs are slower than reads against files, overall query performance was comparable to those for file-based implementation because: 1)Reads are done only for cold start queries and the data is cached in-memory for subsequent operations. 2) Much of the time for query processing is spent in complex operations on in-memory data structures, which are same for both LOB and file-based implementations.

4. Related work

ANSI SQL3[SQL399] defines operators with syntax to create an operator, specify its precedence level, and allows Object methods to be bound to operators. It also defines the rules for transforming the invocation of an operator to the execution of the corresponding Object method. Our support for operators is similar to SQL3 operators. However, the key difference is that the extensible indexing framework allows domain-index based evaluation of the operators.

Our approach of integrating new access methods is similar to the approach described in [Sto86]. However, there are significant differences in the design and implementation of these access methods. [Sto86] requires the access method code to directly use the low-level routines to interface correctly with several DBMS components, including transaction management, crash recovery, buffering and concurrency control. In contrast, with our approach, the new access methods use database constructs such as conventional tables, index-organized tables, or LOBs, to hold the domain index entries. The user-defined functions implementing the access methods use callbacks into the database via SQL interface, for storing, manipulating, and accessing domain index entries. The advantage of this scheme is that the indextype designer does not have to deal with complex database components. Clearly, using SQL, as opposed to low-level interfaces, can cause performance degradation. These performance problems have been addressed by providing batch operations interfaces, and efficient method invocation and execution. The indextype designer can use these interfaces, as well as stored procedures, to minimize round-trips between index processing functions and the server.

The data extenders of IBM DB2 also support user-defined indexing schemes. [Dav98] discusses the functionality of the IBM DB2 spatial extender. Though the paper mentions that user-defined index structures are used to implement extenders, we are not aware of the exact methodology. Microsoft OLE DB [OLE97] index interface abstracts the functionality of B-trees and indexed-sequential files. However, it does not support the functionality required for arbitrary user defined access methods and operator predicates. [Bli99] evaluates the extensibility of the Informix Dynamic server by developing a new indexing scheme.

The generalized search trees [HNP95] provides an index structure for supporting extensible set of queries and data types. However, the focus is on extending the search tree technology to support indexing multiple domains.

5. Limitations and proposed solutions

Although the extensible indexing framework offers a set of advantages, it has some limitations. In this section, we discuss these problems and their possible solutions :

- *User-control over Concurrency and Buffering*: Under the framework a domain index is typically implemented as a collection of database objects, primarily tables and LOBs. The concurrency control and buffering algorithms for domain indexes are restricted to what is available for these database objects. However, users may want different concurrency control and buffering algorithm for domain indexes. For example, consider the case when a file-based index structure is migrated to a LOB in the database, as discussed in Section 3.2.3. Clearly, acquiring a lock on the row containing the LOB may be too prohibitive in terms of concurrency. A possible solution would be to treat the LOB as a page-based store, and use general byte-range locking of LOB bytes to implement appropriate concurrency control algorithms. Also, there is no mechanism to support a different buffer replacement policy for LOB pages that represent a domain index data.
- *Interacting with external data stores*: If the index data is stored outside the database, the transaction manager of the database server does not handle changes to index data. For example, if the user aborts a transaction after making some updates that

also caused domain index updates, the changes to the base table are rolled back whereas changes to the index data are not. We are investigating the use of *database events* [Ora99], which provide a framework to register functions to be invoked when certain database events occur. The indextype designer can register functions for events such as commit and rollback, which contain code to take appropriate actions on index data stored externally.

6. Conclusions

With the extensible indexing framework, users can define domain-specific operators and indexing schemes, and integrate them into Oracle8*i* server. This framework enables domain indexes to operate essentially the same way as any other Oracle built-in index, the primary difference being that Oracle server will invoke user-supplied code specified as part of the indextype to create, drop, truncate, modify, and search a domain index.

By using the extensible indexing framework, users can augment Oracle's built-in indexing capability in several ways. Specifically, new indextypes can be defined to index non-scalar columns such as object types, collection types, and LOBs. The indexing schemes can support predefined built-in relational operators and user-defined operators. In addition, indextypes can be defined to index scalar columns that model domain data, and support processing of domain-specific operations.

A novel aspect of the indexing schemes supported by the framework is the storage of domain index data in one or more Oracle database objects, and use of SQL callbacks to access and manipulate index data from user-supplied index routines. With this approach, the callbacks can exploit the performance and scalability of Oracle SQL without any special effort by the indextype designer. The case studies indicate that the extensible indexing framework is being adopted because it simplifies end-user queries, and/or improves query performance. We believe that this simplicity along with good performance will appeal to an even broader set of customers, and encourage them to integrate domain specific indexing schemes into the database server. Although the framework has limitations when dealing with indexing schemes that require access to low-level database services, it is capable of satisfying indexing requirements for a practical subset of applications.

Acknowledgments

We thank Chin Hong and Anil Nori for their help in defining the framework. Thanks to Alok Srivastava for implementing indextypes and Dinesh Das for implementing optimizer related support. We thank various data cartridge groups for giving us valuable feedback. Thanks also to Sandeepan Banerjee for organizing an effective beta program. We are grateful to Huyen Nguyen and Cheuk Chau for writing exhaustive tests. Special thanks to Jay Banerjee and Vishu Krishnamurthy for their committed support and encouragement throughout the project.

References

[Bli99] Bliujute, R. et al, "Developing a DataBlade for a New Index", *Proceedings of the 15th Data Engineering Conf.*, March 1999.

[Com79] Comer, D. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121-137, June 1979.

[D+95] DeFazio, S. et al., "Integrating IR and RDBMS Using Cooperative Indexing," *Proceedings of the ACM SIGIR Int. Conf. on Information Retrieval*, 1995.

[DOCC99] Daylight/Oracle Chemistry Cartridge, http://www.daylight.com, 1999.

[Gut84] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Structures," *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pp.47-57, June 1984.

[HK95] Hobbs, L., England, K., *Rdb A Comprehensive Guide*, Digital Press, 1995.

[HNP98] Hellerstein, J., Naughton, J., F., Pfeffer, A, "Generalized Search Trees for Database Systems," *Proceedings of the 21th VLDB Conference*, pp.512-522, August 1995.

[Dav98] Davis, J.R, "IBMS's DB2 Spatial Extender: Managing Geo-Spatial Information Within The DBMS," White Paper, Data Management Solutions, IBM, May 1998.

[OLE97] Microsoft OLE DB 1.1 Programmer's Reference and Software Development Kit, Microsoft Press, 1997.

[OIMT99] *Oracle8i interMedia Text Reference, Release 8.1.5*, Oracle Corporation, Part No. A67843-01, February 1999.

[OSpa99] *Oracle8i Spatial User's Guide and Reference, Release 8.1.5*, Oracle Corporation, Part No. A67295-01, February 1999.

[OVIR99] *Oracle8i Visual Information Retrieval User's Guide and Reference, Release 8.1.5*, Oracle Corp., Part No. A67293-01, February 1999.

[ODC99] *Oracle8i Data Cartridge Developer's Guide, Release 8.1.5*, Oracle Corporation, Part No. A68002-01, February 1999.

[Ora99] *Oracle8i Concepts, Release 8.1.5*, Oracle Corporation, Part No. A67781-01, February 1999.

[SQL399] Data Base Language SQL – Part 2: SQL foundation (for SQL3), ISO/IEC DIS 9075-2, 1999.

[Sto86] Stonebraker, M., "Inclusion of New Types in Relational Database Systems", *Proceedings of the IEEE 4th Int. Conf. On Data Engineering*, pp.262-269, February1986.