All Your Data: The Oracle Extensibility Architecture

Sandeepan Banerjee, Vishu Krishnamurthy, Ravi Murthy

Table of contents

Overview

The new Internet computing environment has brought new kinds of data to huge numbers of users across the globe. Multimedia data types like images, maps, video clips, and audio clips were once rarely seen outside of specialty software. Today, many Web-based applications require their database servers to manage such data. Other software solutions need to store data dealing with financial instruments, engineering diagrams, or molecular structures. An increasingly large number of database applications demand content-rich data types and associated content-specific business logic .

With the addition of object-relational extensions, the Oracle8*i* server [OAD99, ODC99] can be enhanced by developers to create their own application-domain-specific data types. For example, one can create new data types representing customers, financial portfolios, photographs or telephone networks – and thus ensure that database programs deal with the same level of abstraction as their corresponding application domain. In many cases, it is desirable to integrate these new domain types as closely as possible with the server so they are treated at par with the built-in types like NUMBER or VARCHAR. With such integration, the database server can be readily extended for new domains.

Oracle8*i* gives application developers greater control over user-defined data types, not only by enabling the capture of domain logic and processes associated with the data, but also by providing control over the manner in which the server stores, retrieves or interprets this data. The Oracle8*i* database contains a series of database extensibility services, which enable the packaging and integration of content-rich domain types and behavior into server-based managed components. Such components are called Data Cartridges [ODC99]. They are developed and managed by means of a set of database interfaces called the Oracle Data Cartridge Interfaces (ODCI). Let us look at the issues involved in creating data cartridges in more detail.

Relational databases, so far, are widely known for efficiently managing and manipulating simple, structured business data. The business data of the early 90s mostly consisted of flat, row oriented data (i.e. tabular data with no nested or structured columns). Also, databases did not maintain unstructured data such as text associated with records, voice clips, or spatial data. With advances in both computer hardware and software technologies, applications are becoming more sophisticated, and they desire efficient integration of heterogeneous, multi-media data sources. For example, it is quite reasonable to store and manipulate data about an employee's salary (structured, relational data) with the employee's resume (textual, non-relational data), and correlate the results with the location of employees on a map. How many employees who know DBMS and make less than $50k live within 50 miles of San Francisco? Surely they deserve raises.

Until recently, the burden on integrating heterogeneous data types and data sources fell on the applications rather than the underlying DBMS. This happened because of three reasons:

1. Databases did not have the capability to store the unstructured or semi-structured data such as freeform text resumes.
2. Databases could not perform specialized querying on high-dimensional data, such as spatial queries on geographic locations.
3. Databases did not provide adequate performance for efficient manipulation of large amounts of content rich data, so that queries such as the above finished in reasonable time.

Specialized applications, therefore, became available from various vendors to provide middle-tiers that perform spatial searches, free text searches etc. However, such loosely integrated specialty middle-tiers have several disadvantages:

- Many functions have to be build repeatedly.
- Applications become too large, too complex, and far too custom-built.
- Even though these mid-tier products can exploit special access and storage methods to manipulate multi-media data, they run outside the DBMS server, causing performance to degrade as interactions with the database server increase.
- Optimizations across data sources cannot be performed efficiently. For instance, a spatial data server knows nothing about text searches and vice versa.
- Each speciality server comes with its own utilities and practices for administering data, causing severe complexity in the backup, restore, and monitor functions necessary to guarantee high availability.

Since processing for content-rich data is beset with problems when done outside the database, the next question that arises is whether databases can support specific rich types inside them. Since it is not clear what constitutes a full set of such types, it seems inefficient to provide, on an ad hoc basis, support for each new type that comes along. The DBMS would have to be re-architected each time a new type is encountered.

In order words, unless all the content-rich types belong to some comprehensive architecture, they will continue to be deviled by issues in re-architecture, cross-type query optimization, uniform programmatic access and so on.

Oracle approached the content-rich data problem from the standpoint of creating such an architecture. Databases must be *extended* to be able to efficiently handle various rich, application-domain-specific data types. Extensibility is the ability to provide support for any user-defined datatype (structured or unstructured) efficiently without having to re-architect the DBMS. Data type support should include: definition of the type, user-defined operations (operators and functions) on the datatype, user-defined storage and access structures for efficient storage and retrieval of the datatype instances, queryability on the datatype instances, etc. Consider the following example:

```
CREATE TABLE patients (
      patient_id  PersonID,
      age         INTEGER,
      medhistory  Text,
      catscan     Image,
      loc         Location
);

CREATE  TABLE cities (
      name  CHAR(20),
      loc   Location,
      population  INTEGER
);
```

Given the above definitions, it should be possible to formulate queries on the different data types in the above tables. for instance: find the number of patients *older than* 50, that *live within 50 miles* of San Francisco, have had a *family medical history of cancer* and there is a *probability greater than 0.6 of finding a tumor* in their CAT scan:

```
SELECT count(p), p.age
FROM  patients p, cities c
WHERE p.age > 50 AND
      c.name = 'San Francisco' AND
      Distance(p.loc, c.loc, 50) AND
      TumorProb(p.catscan) >= 0.6 AND
      Contains(p.medical_history, 'cancer')
GROUP BY p.age;
```

The above example illustrates the queryability we desire of an assortment of multi-media data types. In order to support the above query, it is clear that significant extensions are required to the services normally provided by the DBMS. Among these extensions are:

- user-defined types - the ability to define text, image and location datatypes
- storage of user-defined type instances - the ability to store and manipulate multi-media type instances
- domain-specific operations - support for user-defined functions/operators like Contains(), Distance(), and TumorProb()
- domain-specific indexing - support for indexes specific to text data (to evaluate Contains()), spatial data (Distance()) etc., which can be used to speed the query.
- optimizer extensibility - support for intelligent ordering of query predicates during evaluation. In the above query, it is critical to decide the order in which to evaluate the where-clauses, so that the most restrictive clause can be applied first.The Contains operator evaluation involves text index search; and Distance evaluation involves a spatial index lookup. The most efficient order of evaluation of these different operators and functions depends on the CPU and I/O costs of the respective operations. The TumorProb() function call should be evaluated last if there is no index on it. Since all these operators and functions are user-defined, the optimizer has to be extended to allow type-designers to specify the costs of various operations on the types.

In each case where a service is an extensible one, a interface or API provides access to the service. Figure 1 below shows this extensibility architecture.

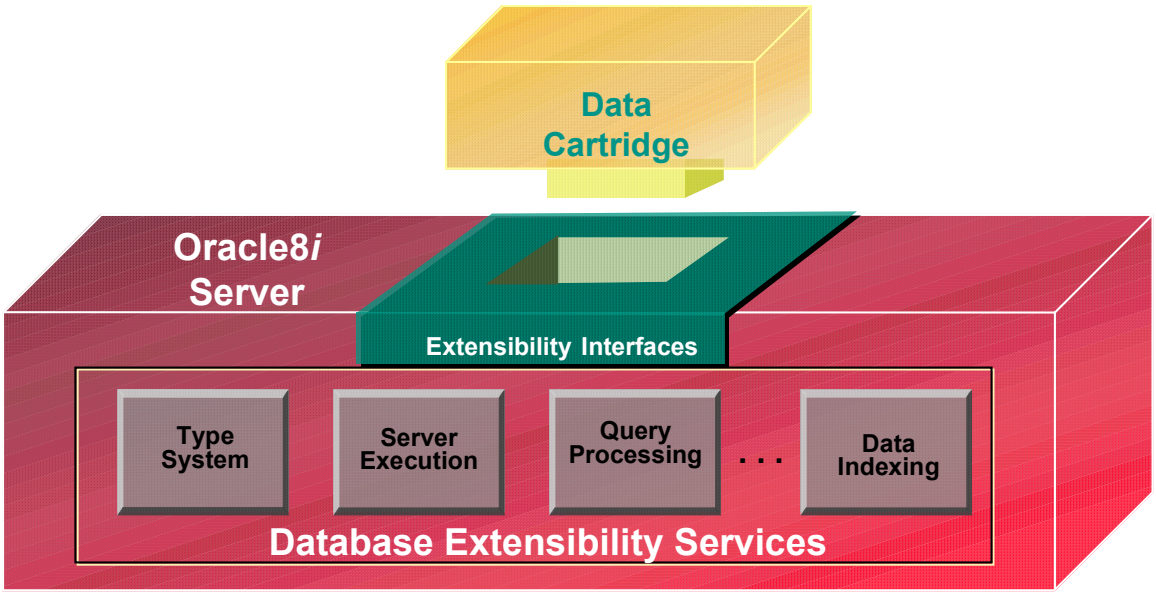

Figure 1: The Oracle Extensibility Architecture

Next, we take a look at the functionality of each extensible service in more detail.

Extensible Type System

The Oracle Type System (OTS) [Ora97, Ora99] provides a high-level (SQL-based) interface for defining types. The behavior for these types can be implemented in Java, C/C++ or PL/ SQL. The DBMS automatically provides the low-level infrastructure services needed for input-output, heterogeneous client-side access for new data types, optimizations for data transfers between the application and the database, and so on. Let us examine the constituents of OTS.

Object Types

An object type, distinct from native SQL data types such as NUMBER, VARCHAR or DATE, is user-defined. It specifies both the underlying persistent data (called 'attributes' of the object type) and the related behavior ('methods' of the object type). Object types are used to extend the modeling capabilities provided by the native data types. They can be used to make better models of complex entities in the real world by binding data attributes to semantic behavior.

There can be one or more attributes in an object type. The attributes of an object type can be the native data types, LOBs, collections, other object types, or REF types (see below).

A method is a procedure or a function that is part of an object type definition. Methods can access and manipulate attributes of the related object type. Methods can be run within the execution environment of the Oracle8*i* Server. In addition, methods can be dispatched to run outside the database as part of the Extensible Server Execution service.

Collection Types

Collections are SQL data types that contain multiple elements. Each element or value for a collection is an instance of the same data type. In Oracle8*i* there are two collection types – VARRAYs and

Nested Tables. A VARRAY contains a variable number of ordered elements. VARRAY data types can be used as a column of a table or as an attribute of an object type. Also, the element type of a VARRAY may be either a native data type such as NUMBER or an object type.

Using Oracle8*i* SQL, a named table type can be created. These can be used as Nested Tables to provide the semantics of an unordered collection. As with VARRAY, a Nested Table type can be used as a column of a table or as an attribute of an object type.

Relationship Types (REF)

It is possible to obtain a reference (or the database pointer) to a standalone instance of an object type. References are important for navigating among object instances, particularly in client-side applications. A special REF operator is used to obtain a reference to a row object.

Large Objects (LOBs)

Oracle8*i* provides the large object (LOB) types to handle the storage demands of images, video clips, documents, and other such forms of data. Large objects are stored in a manner that optimizes space utilization and provides efficient access. More specifically, large objects are composed of locators and the related binary or character data The LOB locators are stored in-line with other table record columns and for internal LOBs (BLOB, CLOB, and NCLOB) the data can reside in a separate storage area. For external LOBs (BFILEs), the data is stored outside the database tablespaces in operating system files. Unlike the Oracle LONG RAW, a table can contain multiple LOB columns. Each such column can be stored in a separate tablespace and even on different secondary storage devices.

There are SQL data definition language (DDL) extensions to create/delete tables and object types that contain large object types. The Oracle8*i* Server provides SQL data manipulation language (DML) commands to INSERT and DELETE complete LOBs. In addition, there is an extensive set of commands for piece-wise reading, writing, and manipulating LOBs via Java, PL/SQL, OLE/DB or the Oracle Call Interface (OCI).

For internal LOB types, the locators and related data participate fully in the transactional model of the Oracle8*i* server. The data for BFILEs does not participate in transactions. However, the BFILE locators themselves are fully supported by server transactions.

With respect to SQL, the data residing within Oracle8*i* LOBs is opaque and not queryable. One can write functions (including methods of object types) to access and manipulate parts of LOBs. In this way the structure and semantics of data residing in large objects can be supplied by application developers.

Opaque Types

The opaque type mechanism provides a way to create new basic types in the database whose internal structure is not known to the DBMS. The internal structure is modeled in some 3GL language (such as C). The database provides storage for the type instances which can be bounded by a certain size with a varying length or of a fixed size. The storage requirement is specified within the type definition. The type methods or functions that access the internal structure are external methods or external procedures in the same 3GL language used to model the structure.

Server Execution Environments

The Oracle8*i* type system de-couples the choice of implementation language for the member method of an object type from its specification. Thus, components of an Oracle8*i* data cartridge can be developed using any of the popular programming languages. In Oracle*8i*, methods, functions, and procedures can be developed using Java, PL/SQL, or external C language routines. Indeed, a type developer can mix and match multiple languages. Thus, the database server runtime environment can be extended by user-defined methods, functions, and procedures.

Java

Java is one of the available choices for server-based execution. Oracle8*i* provides a high performance Java Virtual Machine (JVM) to enable the use of Java in developing stored procedures, object-type methods, standalone functions, and constraints [OJSP99]. This scaleable, multi-user JVM runs in the address space of the database server, and can be used to run standard Java behavior inside the database. There are multiple programming models available with the JVM. JDBC allows object-relational statement-wise access to data [OJDBC99]. SQLJ, a standard precompiler technology, allows SQL to be embedded directly into Java code [OSQLJ99]. Associated with the JVM is a server-based Object Request Broker (ORB), which enables a Enterprise JavaBeans (EJB) programming model [OEJB99]. The server ORB is fully compliant with the Common Object Request Broker (CORBA) specification. Finally, it is possible to perform ahead-of-time compilation on server-based Java code, so that the costs of interpreting this code is not taken at each invocation. Such 'native compilation' capabilities make it possible to write computationally intensive data cartridge behavior in Java.

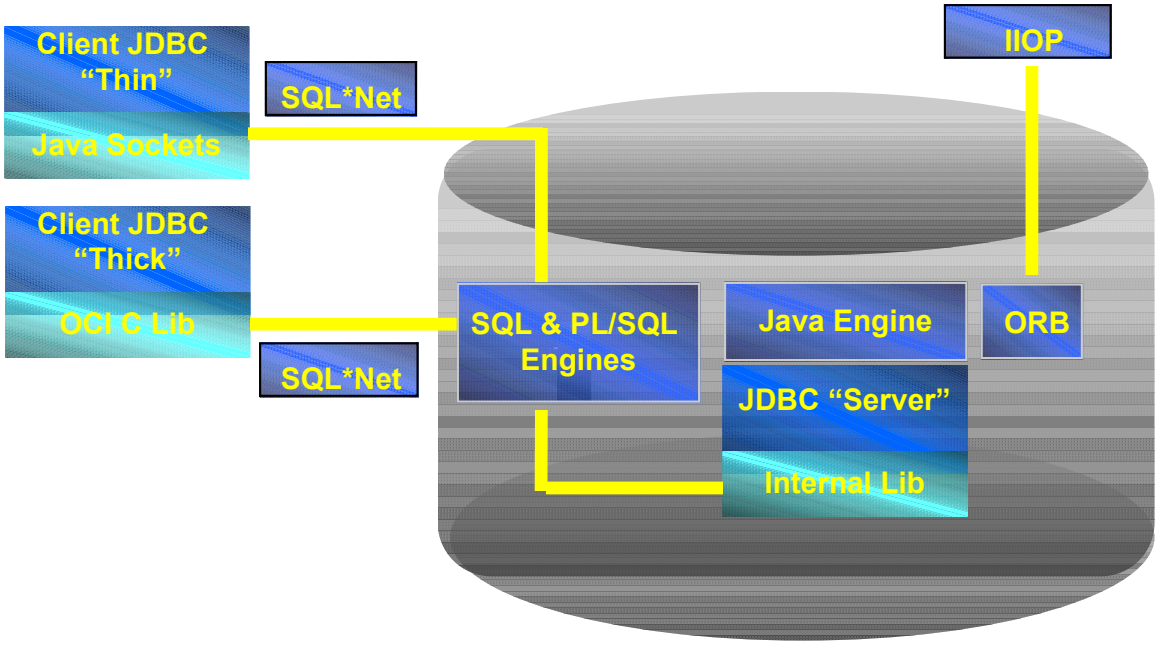

Figure 2: Java and Oracle8*i*

In addition to server-based Java execution, Oracle8*i* also provides client-side Java access to the JVM, as well as the SQL and PL/SQL engines by means of JDBC. Figure 2 shows how these fit schematically in the overall database architecture.

PL/SQL

In Oracle8*i*, PL/SQL offers a data cartridge developer a powerful procedural language that supports all the object extensions for SQL [PLSQL99]. With PL/SQL, program logic can execute on the server performing traditional procedural language operations such as loops, if-then-else clauses, and array access. All of this processing occurs in a transactional SQL environment where DML statements can be executed to retrieve and modify object data.

C/C++

While PL/SQL and Java are comprehensive languages, certain computations such as a Fast Fourier Transform or an image format conversion are handled more efficiently by C programs. With the Oracle8*i* Server, C language programs can be called from PL/SQL. As shown in the figure below, external programs are executed in a separate address space from the server. This ensures that the database server is insulated from any program failures that might occur in external procedures and under no circumstances can an Oracle database be corrupted by such failures.

In general, the Extensible Server Execution Environment enables an external C routine to be used wherever a PL/SQL subprogram could be called – such as the body of a PL/SQL method for an object type, a database trigger or a PL/SQL function embedded in an SQL statement. Figure 3 shows the process of dispatching an external routine.

External routines need not be confined to C; in fact, any external language that is capable of generating a dynamically linked library or shared object file can be used to implement behavior in the Extensible Server Execution service. C++, Pascal etc. are all available as implementation choices.

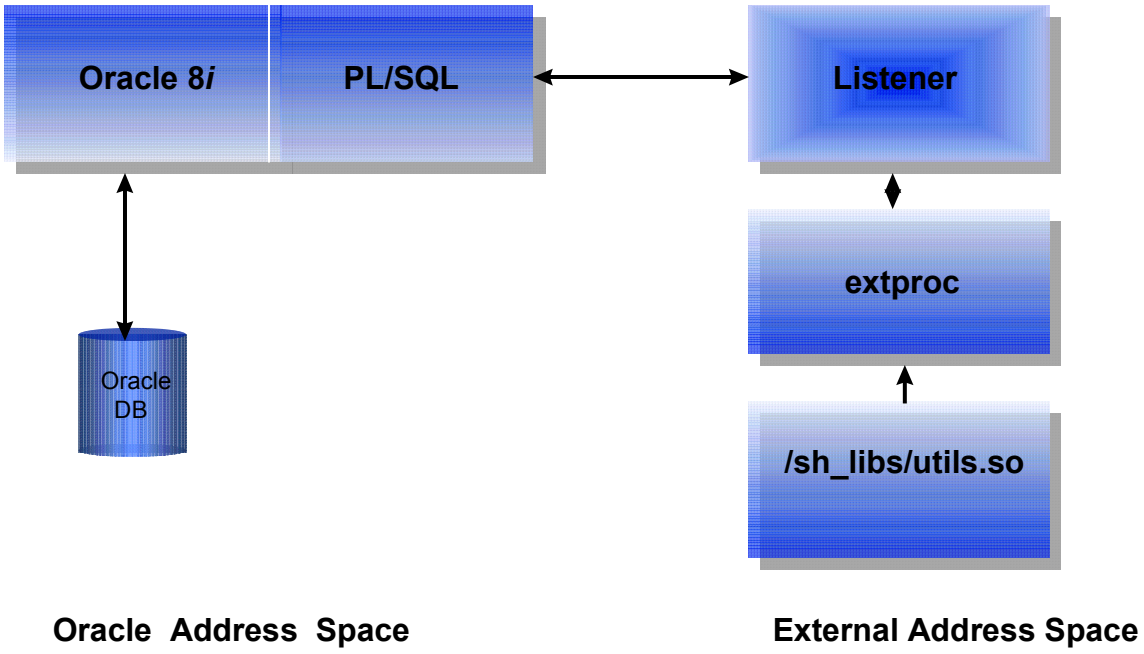

Figure 3: Dispatching External Routines

With certain reasonable restrictions, external routines can *call back* to the Oracle server using OCI. Callbacks are particularly useful for processing LOBs. For example, by using callbacks an external routine can perform piece-wise reads or writes of LOBs stored in the database. External routines can also use callbacks to manipulate domain indexes stored as index-organized tables (see the section on extensible indexing below) in the database. External routines can be procedures or functions (the difference is that functions have a return value and procedures do not.)

Safe Execution

Opening up the server execution environment creates a new problem for component databases. As long as all the operating parts of a database came from one vendor, safety and operational reliability of the data could be achieved relatively easily. As database evolve into platforms for hosting specialized data and behavior, the end-user could see reliability reduced as the least reliable vendor-component becomes the weakest link in the chain.

One key characteristic of the Oracle Extensibility architecture is that it is always safe for an end-user to run cartridges created by third-parties running on the Oracle platform. PL/SQL and Java are interpreted and therefore safe to run in the server's address-space. Even in the case where Java code is compiled, the native compiler performs various bounds checking operations to ensure that the generated compilable code is safe to run. 'Unsafe' cartridge code – written in C or C++ -- is run outside the address space of the database using the so-called "extproc" mechanism.

Extensible Indexing

Typical database management systems support a few types of access methods (for example B+trees, hash indexes) on some set of data types (numbers, strings, etc.). In recent years, databases have been used to store different types of data like text, spatial, image, video, and audio. In these complex domains, there is a need for indexing complex data types and specialized indexing techniques. For simple data types such as integers and small strings, all aspects of indexing can be easily handled by the database system. This is not the case for documents, images, video clips, and other complex data types that require content-based retrieval. Complex data types have application specific formats, indexing requirements, and selection predicates. For example, there are many different document encodings (for example ODA, SGML, plain text) and information retrieval techniques (for example keyword, full-text boolean, similarity, and probabilistic). Similarly, R-trees are an efficient method of indexing spatial data. No database server can be built with support for all possible kinds of complex data and indexing. Oracle's solution is to build an extensible server which provides the ability to the application developer to define new index types [ODC99].

The framework to develop new index types is based on the concept of cooperative indexing where a data cartridge and the Oracle server cooperate to build and maintain indexes for data types including text, spatial, and Online Analytical Processing (OLAP). The cartridge is responsible for defining the index structure, maintaining the index content during load and update operations, and searching the index during query processing. The index structure itself can either be stored in the Oracle database (e.g. in heap or Index-Organized Tables) or externally (e.g. in operating system files). However, it is highly desirable for reasons of concurrency control and recovery, to have the physical storage of domain indexes within the Oracle database.

To this end, Oracle8*i* introduces the concept of an Indextype. The purpose of an Indextype is to enable efficient search and retrieval functions for complex domains such as text, spatial, image, and OLAP. An Indextype is analogous to the sorted or bit-mapped index types that are built in the Oracle server. The difference is that the routines implementing an Indextype are provided by the cartridge developer, whereas the Oracle server kernel implements the built-in indexes. Once a new Indextype has been implemented by a data cartridge developer, end users of the data cartridge can use it just like the built-in index types.

With Extensible Indexing, the application:

- Defines the structure of the domain index as a new Indextype;
- Stores the index data either inside the Oracle database (in the form of tables) or outside the Oracle database;
- Manages, retrieves, and uses the index data to evaluate user queries.

When the database server handles the physical storage of domain indexes, cartridges must be able to:

- Define the format and content of an index. This enables cartridges to define an index structure that can accommodate a complex data object.
- Build, delete, and update a domain index. The cartridge handles building and maintenance of the index structures. Note that this is a significant departure from the "automatic" indexing features provided for simple SQL data types. Also, since an index is modeled as a collection of tuples, in-place updating is directly supported.

- Access and interpret the content of an index. This capability enables the data cartridge to become an integral component of query processing. That is, the content-related clauses for database queries are handled by the data cartridge.

Typical database systems (relational and object-relational ones) do not support extensible indexing. Many applications maintain file-based indexes for complex data residing in relational 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 extensible indexes, the Oracle8*i* server significantly reduces the level of effort needed to develop solutions involving high-performance access to complex data types. The table below lists the functionality of the ODCIIndex interface.

	Interface Routine	**Description**
Index Create	ODCIIndexCreate	Creates the domain index according to user-specified parameters
Index Drop	ODCIIndexDrop	Drops the domain index
Index Scan	ODCIIndexStart	Initializes the scan of the domain index
	ODCIIndexFetch	Fetches from the domain index: returns the ROWID of each successive row satisfying the operator predicate
	ODCIIndexClose	Ends the current use of the index
Insert	ODCIIndexInsert	Maintains the domain index structure when a row is inserted in the indexed table
Delete	ODCIIndexDelete	Maintains the domain index structure when a row is deleted
Update	ODCIIndexUpdate	Maintains the domain index structure when a row is updated
Truncate	ODCIIndexTruncate	Deletes the domain index data preserving the structure
Alter	ODCIIndexAlter	Alters the domain index
Meta Data	ODCIIndexGetMeta Data	Allow import/export of domain-index-specific metadata

Table 1 : ODCIIndex Interface

Before we provide an example of Extensible Indexing, it is necessary to introduce a few additional constructs, all necessary to add power and flexibility to the database indexing service. We discuss Index-Organized Tables and Function-Based Indexes below, followed by a discussion of User-defined operators, before presenting the example for Extensible indexing.

Index-Organized Tables

An index-organized table (IOT) is a useful tool in the armory of a cartridge developer using extensible indexing. An IOT differs from an ordinary table in that the data for the table is held in its associated index. Changes to the table data, such as adding new rows, updating rows, or deleting rows, result only in updating the index.

The index-organized table is like an ordinary table with an index on one or more of its columns, but instead of maintaining two separate storages for the table and the B*-tree index, the database system only maintains a single B*-tree index which contains both the encoded key value and the associated column values for the corresponding row. Rather than having a row's rowid as the second element of the index entry, the actual data row is stored in the B*-tree index. The data rows are built on the primary key for the table, and each B*-tree index entry contains pairs <primary_key_value, non_primary_key_column_values>.

IOTs are suitable for accessing data by the primary key or any key that is a valid prefix of the primary key. There is no duplication of key values because only non-key column values are stored with the key. One can build secondary indexes to provide efficient access to other columns. Applications manipulate the IOT just like an ordinary table, using SQL statements. However, the database system performs all operations by manipulating the corresponding B*-tree index. Table 2 summarizes the main distinctions between access of ordinary tables and IOTs.

IOTs can be very useful to developers of domain indexes in that they provide a 'canned' B*-tree index to store their data.

Ordinary Table	Index-Organized Table
Rowid based access	Primary key based access
Physical rowid in ROWID pseudocolumn allows building secondary indexes	Logical rowid in ROWID pseudocolumn allows building secondary indexes
Rowid uniquely identifies a row; primary key can be optionally specified	Primary key uniquely identifies a row; primary key must be specified
Sequential scan returns all rows	Full-index scan returns all rows in primary key order
UNIQUE constraint and triggers allowed	UNIQUE constraint not allowed, triggers allowed
Can be stored in a cluster with other tables	Cannot be stored in a cluster

Table 2 : Ordinary Tables vs. Index-Organized Table

Function-based Indexing

So far, we have discussed indexing data in various ways. Another intriguing possibility open to Data Cartridge developers is the ability to index on behavior.

To address efficient evaluation of a query when the predicate is based on an object method, Oracle8*i* supports *function-based indexes*. Users can create indexes on functions (object methods) and expressions that involve one or more columns in the table being indexed. A function-based index precomputes the value of the function or expression and stores it in the index. Function-based index are created as either B*-tree or bitmap index. The function used for building the index can be an arithmetic expression or an expression that contains an object type method or a standalone SQL function.

Function-based indexes provide an efficient mechanism for evaluating SQL statements that contain functions in their WHERE clauses. One can create a function-based index to materialize computational-intensive expressions in the index, so that Oracle does not need to compute the value of the expression when processing SELECT or DELETE statements. However, when processing INSERT and UPDATE statements, Oracle must still evaluate the function to process the statement.

For example, suppose a table contains all purchase order objects, and suppose TotalValue is a method defined for purchase_order type that returned the total value of a purchase order object by summing up the values of the individual line items of the purchase order. Then the following index:

```
CREATE INDEX TotalValueIndx ON purchase_order_table p
p.TotalValue();
```

can be used instead of evaluating the TotalValue method, e.g., when processing queries such as this:

```
SELECT p.order_id FROM purchase_order_table_p WHERE
p.TotalValue() >10000;
```

The ability to build functional indexes thus extends the database indexing service in a fundamental way.

User-defined Operators

Data cartridge developers find it useful to define domain-specific operators and integrate them into the Oracle8*i* server along with extensible indexing schemes that such operators take advantage of while accessing data. The ability to increase the semantics of the query language by adding such domain-specific operators is akin to extending the query service of the database.

Oracle8*i* 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.

Oracle8*i* allows users to extend the set of operators by defining new ones with user specified behavior. Like built-in operators, they take a set of operands as input and return a result. The implementation of the operator 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 and returns TRUE if the document contains the specified keyword, we can write an SQL query as :

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

Oracle8*i* uses indexes to efficiently evaluate some built-in operators — for example, a B-tree index can be used to evaluate the comparison operators =, > and <. Similarly, in Oracle8*i*, user-defined domain indexes can be used to efficiently evaluate user-defined operators.

In general, user-defined operators are bound to functions. However, operators can also be evaluated using indexes. For instance, the equality operator can be evaluated using a hash index. An indextype provides index-based implementation for the operators listed in the indextype definition.

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. The Oracle8*i* server executes the function when the operator is invoked. Multiple operator bindings can be defined as long as they differ in their signatures. Thus, any operator can have an associated set of zero or more bindings. Each of these bindings can be evaluated using a user-defined function which could be one of the following:

- Stand-Alone Functions
- Package Functions
- Object Member Methods

User-defined operators can be invoked anywhere built-in operators can be used — that is, wherever expressions can occur. For example, user-defined operators can be used in the following :

- select-list of a select command
- condition of a where clause
- order by and group by clauses

When an operator is invoked, its evaluation is transformed into the execution of one of the functions bound to it. This transformation is based on the data types of the arguments to the operator. If none of the functions bound to the operator satisfy the signature with which the operator is invoked, an error occurs. There might be some implicit type conversions present during the transformation process.

The following example illustrates the extensible indexing and user-defined operator framework.

Example: Extensible Indexing and Operators

Consider a text retrieval application. For such applications, indexing involves parsing the text and inserting the words or tokens into an inverted index. Such index entries typically have the following logical form

```
(token, <docid, data>)
```

where token is a word or stem that is a term in searches, docid is a unique identifier for a document this word occurs in, and data is a segment containing information on the how many times or where in the document the word occurs.

A sample index entry for such an application would look like:

(Ulysses, <5, 3, [7 62 225]>, <26, 2, [33, 49]>, ...)

In this sample index entry, the token *Ulysses* appears in document 5 at 3 locations (7, 62, and 225) and in document 26 at 2 locations (33 and 49). Note that the index would contain one entry for every document with the word *Ulysses*.

Defining a Text Indexing Scheme

The sequence of steps required to define a text indexing scheme using a text Indextype are:

- Define and code functions to support functional implementation of operators which eventually would be supported by the text indextype.

Suppose our text indexing scheme is in the context of a text data cartridge that intends to support an operator *Contains*. The operator *Contains* takes as parameters a text value and a key and returns a boolean value indicating whether the text contains the key. The functional implementation of this operator is a regular function defined as :

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

- Create a new operator and define its specification, namely the argument and return data types, and the functional implementation:

```
CREATE OPERATOR Contains
BINDING (VARCHAR2, VARCHAR2) RETURN BOOLEAN
USING TextContains;
```

- Define a type or package that implements ODCIIndex. This involves implementing routines for index definition, index maintenance, and index scan operations.

The index definition routines (ODCIIndexCreate, ODCIIndexAlter, ODCIIndexDrop, ODCIIndexTruncate) build the text index when the index is created, alter the index information when the index is altered, remove the index information when the index is dropped, and truncate the text index when the base table is truncated.

The index maintenance routines (ODCIIndexInsert, ODCIIndexDelete, ODCIIndexUpdate) maintain the text index when the table rows are inserted, deleted or updated.

The index scan routines (ODCIIndexStart, ODCIIndexFetch, ODCIIndexClose) implement access to the text index to retrieve rows of the base table that satisfy the operator predicate. In this case, Contains(...) forms a boolean predicate whose arguments are passed in to the index scan routines. The index scan routines scan the text index and return the qualifying rows to the system.

```
CREATE TYPE TextIndexMethods (
       FUNCTION ODCIIndexCreate(...)
       ...
);

CREATE TYPE BODY TextIndexMethods  (
       ...
```

```
);
```

- Create the Text Indextype schema object. The Indextype definition also specifies all the operators supported by the new indextype and the type that implements the index interface.

```
CREATE INDEXTYPE TextIndexType
FOR Contains(VARCHAR2, VARCHAR2)
USING TextIndexMethods;
```

Using the Text Indexing Scheme

Suppose that the text Indextype presented in the previous section has been defined in the system. The user can define text indexes on text columns and use the associated Contains operator to query text data. Further, suppose an Employees table is defined as follows:

```
CREATE TABLE Employees (name VARCHAR2(64), id INTEGER,
            resume VARCHAR2(2000));
```

A text domain index can be built on the resume column as follows:

```
CREATE INDEX ResumeIndex ON Employees(resume)
INDEXTYPE IS TextIndex;
```

The text data in the resume column can be queried as:

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

The query execution will use the text index on resume to efficiently evaluate the Contains() predicate.

Extensible Optimizer

The extensible optimizer functionality enables authors of user-defined functions and indexes to create statistics collection, selectivity, and cost functions [ODC99]. This information is used by the optimizer in choosing a query plan. The cost-based optimizer is thus extended to use the user-supplied information.

The optimizer generates an execution plan for a SQL statement (for simplicity, consider a SELECT statement — the same applies for other statements). An execution plan includes an access method for each table in the FROM clause, and an ordering (called the join order) of the tables in the FROM clause. System-defined access methods include indexes, hash clusters, and table scans. The optimizer chooses a plan by generating a set of join orders or permutations, computing the cost of each, and selecting the one with the lowest cost. For each table in the join order, the optimizer computes the cost of each possible access and join method choosing the one with the lowest cost. The cost of the join order is the sum of the access method and join method costs. The costs are calculated using algorithms which together compose the cost model. A cost model can include varying level of detail about the physical environment in which the query is executed. Oracle's present cost model includes only the number of disk accesses with minor adjustments to compensate for the lack of detail. The

optimizer uses statistics about the objects referenced in the query to compute the costs. The statistics are gathered using the ANALYZE command. The optimizer uses these statistics to calculate cost and selectivity. The selectivity of a predicate is the fraction of rows in a table that will be chosen by the predicate.

Extensible indexing functionality enables users to define new operators, index types, and domain indexes. For such user-defined operators and domain indexes, the extensible optimizer gives data cartridge developers control over the three main components used by the optimizer to select an execution plan: statistics, selectivity, and cost. We look at each of these components in more detail.

Statistics

The ANALYZE command is extended so that whenever a domain index is to be analyzed, a call is made to the cartridge-specified statistics collection function. The representation and meaning of these user-collected statistics is not known to the database.

In addition to domain indexes, cartridge-defined statistics collection functions are also supported for individual columns of a table and data types (whether built-in types or object types). In the former case, whenever a column is analyzed, in addition to the standard statistics collected by the database, the user-defined statistics collection function is called to collect additional statistics. If a statistics collection function exists for a data type, it is called for each column of the table being analyzed of the specified type. For example, the following statement associates a statistics type ImgStats_t which implements the statistics collection function (ODCIStatsCollect) with the image column imgcol of the table tab.

```
ASSOCIATE STATISTICS WITH COLUMNS tab.imgcol USING ImgStats_t;
```

The following ANALYZE command collects statistics for the tab table. In addition to the usual statistics collected by Oracle, the ODCIStatsCollect function is invoked to collect extra statistics for the imgcol column.

```
ANALYZE TABLE tab COMPUTE STATISTICS;
```

Example: Statistics

Consider images stored in a BLOB column within a table created as :

```
CREATE TABLE ImgTab (
      name VARCHAR2(100),
      imgcol BLOB
);
```

The following statement associates a statistics type ImgStats_t which implements the statistics collection function (ODCIStatsCollect) with the image column imgcol of the table tab.

```
ASSOCIATE STATISTICS WITH COLUMNS ImgTab.imgcol USING
ImgStats_t;
```

The type ImgStats_t implements the ODCIStats interface.

```
CREATE TYPE ImgStats_t AS OBJECT
```

```
(
      STATIC FUNCTION ODCIStatsCollect(…)…
      …
);
```

The following ANALYZE command collects statistics for the ImgTab table. In addition to the usual statistics collected by Oracle, the ODCIStatsCollect function is invoked to collect extra statistics for the imgcol column. It could collect any relevant information regarding the images. For example, the average size of the images is a useful indicator of the time required to process the images.

```
ANALYZE TABLE ImgTab COMPUTE STATISTICS;
```

Selectivity

The optimizer uses statistics to calculate the selectivity of predicates. The selectivity is the fraction of rows in a table that will be chosen by the predicate and is a number between 0 and 1. The selectivity of a predicate is used to estimate the cost of a particular access method. It is also used to determine the optimal join order. A poor choice of join order by the optimizer could result in a very expensive execution plan.

By default, the optimizer uses a standard algorithm to estimate the selectivity of selection and join predicates. However, the algorithm does not work very well when predicates contain functions or type methods. Additionally, in Oracle8*i* predicates can contain user-defined operators about which the optimizer does not have any information and so cannot compute an accurate selectivity.

For greater control over the optimizer's selectivity estimation, the extensible optimizer enables data cartridge developers to specify user-defined selectivity functions for predicates containing user-defined operators, standalone functions, package functions or type methods. The user-defined selectivity function will be called by the optimizer whenever it encounters a predicate with one of the following forms:

op(...) relop <constant>

<constant> relop op(...)

op(...) LIKE <constant>

where *op(...)* is a user-defined operator, standalone function, package function or type method, *relop* is any of the standard comparison operators (<, <=, =, >=, >), and *<constant>* is a constant value expression or bind variable. For such cases, data cartridges can define selectivity functions that will be associated with *op(...)*. The arguments to *op* can be columns, constants, bind variables or attribute references. When such a predicate is encountered, the optimizer will call the user-defined selectivity function and pass the entire predicate as an argument including the operator, function or type method and its arguments, the relational operator *relop*, and the constant expression or bind variable. The return value of the user-defined selectivity function must be between 0 and 1, inclusive; values outside this range are ignored by the optimizer. Typically, the arguments and the statistics collected are used to estimate the selectivity of an operation.

Example: Selectivity

Consider a function Brightness() defined on images that returns a value between 0 and 100 to indicate the level of brightness. A selectivity function can be associated by implementing the ODCIStatsSelectivity function within a type, say ImgStats_t and executing the following statement :

```
ASSOCIATE STATISTICS WITH FUNCTIONS Brightness
USING ImgStats_t;
```

Now, if a user executes a query of the form :

```
SELECT * FROM ImgTab
WHERE Brightness(imgcol) BETWEEN 50 and 60;
```

the selectivity of the predicate is computed by invoking the user supplied implementation of the ODCIStatsSelectivity function.

Cost

The optimizer estimates the cost of various access paths to choose an optimal plan. For example, it may compute the cost of using an index and a full table scan in order to be able choose between the two. However, for data cartridge defined domain indexes, the optimizer does not know the internal storage structure of the index. Thus, the optimizer cannot make a good estimate of the cost of using such an index. Similarly, the default optimizer model assumes that the cost of I/O dominates – and that other activities like function evaluations have zero cost. This is only true when functions include relatively inexpensive built-in functions. User-defined functions can be very expensive since they can be CPU-intensive. User-defined functions can invoke recursive SQL. When the function argument is a file LOB, there may be substantial I/O cost.

For superior optimization, the cost model is extended to enable users to define costs for domain indexes, user-defined functions, standard standalone functions, package functions, and type methods. The user-defined costs can be in the form of default costs that the optimizer simply looks up, or can be full-blown cost functions which the optimizer calls to compute the cost.

User-defined cost, like user-defined selectivity, is optional on the part of a data cartridge. If no user-defined cost is available, the optimizer uses its internal heuristics to compute an estimate. However, in the absence of useful information about the storage structures in user-defined domain indexes and functions, such estimates can be very inaccurate and may result in the choice of a sub-optimal execution plan.

User-defined cost functions for domain indexes are called by the optimizer only if a domain index is a valid access path for a user-defined operator.

User-defined cost functions can return three parameters. Each parameter represents the cost of a single execution of a function or domain index implementation:

1. *cpu* - Number of machine instructions executed by the function or domain index implementation.
2. *i/o* - Number of data blocks read by the function or domain index implementation.

3. *network* - Number of data blocks transmitted. This is valid for distributed queries as well as functions and domain index implementations.

Example: Cost

Consider again the query involving the Brightness() function introduced in the previous section.

```
SELECT * FROM ImgTab
WHERE Brightness(imgcol) BETWEEN 50 and 60;
```

The optimizer will invoke the ODCIStatsFunctionCost() function implemented within ImgStats_t to estimate the cost of executing the Brightness() function. Typically, the cost function retrieves the user collected statistics - e.g. in this case, average length of the image column - and computes the cost of one invocation of the function.

The table below describes the ODCIStats interface.

	Interface Routine	Description
Statistics	ODCIStatsCollect	Collects statistics for column and index data.
	ODCIStatsDelete	Drops statistics.
Selectivity	ODCIStatsSelectivity	Estimates the selectivity of a predicate involving user defined functions or operators.
Cost	ODCIStatsFunctionCost	Accepts information about the function parameters and computes the cost of a single execution.
	ODCIStatsIndexCost	Accepts information about the operator predicate and computes the cost of the domain index scan.

Table 3 : ODCIStats Interface

User-defined Aggregates

Typical database systems support a small number of aggregate operators (e.g. MAX, MIN) over scalar datatypes such as number and character strings. However, it is desirable to provide the capability of adding new aggregate operators to the DBMS. For instance, a new aggregate operator SecondMax() - which ignores the highest value and returns the second highest - might be necessary in some application. Further, in complex domains, the semantics of aggregation is not known to the DBMS and has to be provided by the domain code. For instance, MAX() in the spatial domain may refer to the geometry with the largest enclosed area.

User Defined Aggregate operators (UDAG) are the mechanism to incorporate new aggregate operators with user specified aggregation semantics. Users can specify a set of routines that implement the aggregation logic. Once the aggregate operator is registered with Oracle, it can be used by users wherever built-in aggregates can be used.

An aggregate operator operates over a set of rows and returns a single value. The sets of rows for aggregation are typically identified using a GROUP BY clause. For example:

```
SELECT SUM(T.Sales)FROM AnnualSales T
GROUP BY T.State
```

Conceptually, an aggregate value is computed in three steps. Taking the example of SUM(), the steps are

- Initialize : intialize the computation.

  Action - assign 0 to runningSum and runningCount variables.

- Iterate : iteratively examine each of the tuples, and perform necessary computations.

  Action - add input number value to runningSum variable, and increment runningCount variable by 1

- Terminate : Compute the resulting value.

  Action - Compute the Average as (runningSum/runningCount)

New aggregate operators can be registered by providing the implementations for the above aggregation steps.

The variables runningSum and runningCount, in the above example, determine the *state* of the aggregation. We can think of the state as a *state variable*, an object that contains runningSum and runningCount as elements. Thus, the Initialize function initializes the state variable, Iterate updates it and Terminate uses the state variable to return the resultant value. Note that the state variable completely determines the state of the aggregation.

Thus, with user-defined aggregate operators,

1. The application defines the set of implementations of the UDAG, and specifies each of the implementations in terms of the implementation routines, and the data types of the argument values and return value.
2. The application creates the implementation routines in C++ or Java.
3. The application uses the UDAG in SQL query statements.

Example: User-defined Aggregates

This section presents an example of a user-defined aggregate operator. An application requires the use of a TopTen aggregate operator which has the following behaviour : The aggregate operator computed over a set of values determines the ten largest values, and returns a single object having the ten values as its components.

The end-user would like to use the TopTen operator in queries like the following :

```
SELECT TopTen(A1.DollarSales) FROM AnnualSales A1
```

```
GROUP BY A1.State
```

The TopTen aggregate operator is implemented in the following steps:

- Specify the aggregate operator in terms of its implementations, and specify the data types of the argument values, and return value for each implementation as follows :

```
CREATE OPERATOR TopTen AS AGGREGATE
BINDING (NUMBER) RETURN TenNUMBER USING TopTenNumber;
```

TenNUMBER is a collection type capable of holding ten values and is the return type of the aggregate operator implementation.

- Implement the aggregate operator implementation routines in any language supported by Oracle for type methods e.g. PL/SQL, C,C++ or Java. The implementation routines for aggregation are member methods of the TopTenNumber object type.

```
CREATE TYPE TopTenNumber
(
      FUNCTION ODCIAggregateInitialize(...)
      ...
);
CREATE TYPE BODY TopTenNumber
(
      FUNCTION ODCIAggregateInitialize() AS
      BEGIN
      ......
      END ODCIAggregateInitialize;
...
);
```

The table below describes the ODCIAggregate interface.

Action	Interface Routines	Description
Serial Aggregation	ODCIAggregateInitialize	Initializes aggregation context
	ODCIAggregateIterate	Accepts next batch of rows and updates aggregation context
	ODCIAggregateTerminate	Returns aggregate value
Parallel Aggregation	ODCIAggregateParInit	Initializes parallel aggregation context
	ODCIAggregateParIter	Accepts next batch of rows and updates the parallel aggregation context
	ODCIAggregateParTerm	Returns the final parallel aggregation context
	ODCIAggregateSuperAggr	Accepts set of aggregation context and returns aggregate value

Table 4 : ODCIAggregate Interface

Using the User-defined Aggregates

User-defined aggregate operators can be used in DML statements similar to built-in aggregates. The evaluation of the UDAG triggers the invocation of the underlying user-supplied routines to compute the aggregate value. For example,

```
SELECT TopTen(A1.DollarSales) FROM AnnualSales A1
GROUP BY A1.State
```

triggers the invocation of the initialization routine followed by one or more invocations of the iteration routine followed by a invocation of the termination routine.

Abstract Tables

A cartridge developer might occasionally need to access data that is outside any database. Such data may have a certain structure, albeit different from the structure of relational or object-relational databases. For example, some data may be stored as XML files on a file system. As part of data cartridge operations, it is sometimes important to perform composite operations that span such external data as well as internal, tabular data. The best way to combine such different structures is by providing cartridge developers the ability to create a uniform table metaphor over all data by constructing 'abstract' tables corresponding to the external sources.

An abstract table is a "virtual" table where the table data is retrieved by invoking user registered functions. In its simplest form, the user implements iteration routines to scan the rows of the table. However, abstract tables provide a firm framework for supporting user defined data manipulation as well. The user can implement routines that will be invoked when the abstract table is created, dropped and when rows are inserted/deleted/updated.

Abstract tables also support other table features such as building secondary indexes, defining constraints and triggers.

An abstract table is created in a manner similar to regular tables, the difference being that the name of the underlying implementation type is specified. The implementation type contains the implementations of the ODCITable interface - consisting of the table scan and manipulation routines. The ODCITable interface is listed after the example.

Example: Abstract Tables

Suppose we want to read Employee data that originates from a set of operating-system files.

```
CREATE ABSTRACT TABLE emps_filetab
(
      name VARCHAR2(30),
      id NUMBER,
      mgr VARCHAR2(30)
)
USING LoaderFileReader
PARAMETERS ('/tmp/emp.dat');
```

The abstract table definition of *emps_filetab* specifies the name of the object type *LoaderFileReader* that implements the ODCITable interface routines. The ODCITable interface includes the create/drop, DML and scan routines. The PARAMETERS clause can be used to pass user defined parameters to the ODCITableCreate routine - in this example, the path to the loader file is specified.

Once the abstract table has been created, it can be used exactly like regular tables eg. creating indexes, performing DML operations, queries, etc.

The ODCI Table interface contains the definitions of all the routines that need to implemented by the abstract table implementor. Note that the definitions are static and are specified by Oracle - but their actual implementations need to be provided by the user. The ODCITable interface consists of several classes of routines

- Query Routines - These are the set of routines that are invoked by Oracle while querying abstract table data. These are also the same set of routines that are implemented for table functions.
- DML Routines - These are the set of routines that are invoked by Oracle to insert/delete/update rows of the abstract table.
- DDL Routines - These are the set of routines that are invoked by Oracle when an abstract table is created or dropped.

The implementations of all the ODCITable interface routines are provided by the user in the form of an object type. for example, the type LoaderFileReader contains the implementation of the DDL and query routines for the emps_filetab abstract table.

```
CREATE TYPE LoaderFileReader AS OBJECT
(
      MEMBER FUNCTION ODCITableStart(...) RETURN NUMBER;
      MEMBER FUNCTION ODCITableFetch(...) RETURN NUMBER;
      MEMBER FUNCTION ODCITableClose(...) RETURN NUMBER;
);
CREATE TYPE BODY LoaderFileReader AS
...
END;
```

	Interface Routine	**Description**
Table Scan	ODCITableStart	Initializes a full scan of an abstract table.
	ODCITableFetch	Accepts the scan context and returns the next set of rows.
	ODCITableClose	Performs cleanup at the end of scan.
Rowid Access	ODCITableLookup	Retrieves row corresponding to the specified row identifier.
Describe Row	ODCITableDescribe	Returns the metadata descriptor of a row.
Query Rewrite	ODCITableRewrite	Returns a SQL query string that can be plugged into original query in place of the abstract table.

Data Manipulation	ODCITableInsert	Insert a new row into the table.
	ODCITableDelete	Delete a row from the table.
	ODCITableUpdate	Update a row of the table.
Data Definition	ODCITableCreate	Processes parameters specified while creating the abstract table.
	ODCITableDrop	Performs cleanup when abstract table dropped.

Table 5 : ODCITable Interface

When the user executes a query over the abstract table, a scan is set up to iterate over the rows of the table.

SELECT * FROM emps_filetab;

Table Functions

Data cartridge developers may also need dynamic, iterative behavior on virtually created tables. The extensibility architecture also provides iterative Table Functions to complement Abstract Tables.

There are scenarios when data is inherently dynamic and depends on user-supplied parameters. Such data cannot be modeled easily using Abstract Tables. For example, one might want to access data present in an external web site (identified by a URL). A function ReadURL() can be implemented to take in the URL as input and return a collection representing the read data.

```
SELECT * FROM TABLE(ReadURL("www.oracle.com"));
```

This solution has some drawbacks. The entire result set is returned from ReadURL() as a single collection. This not only affects the response time of the query but also consumes more resources. Table functions are the right answer to this problem. They are similar to regular functions returning collections, except that the result is returned iteratively (i.e. in subsets). This is accomplished by implementing the table scan routines within the ODCITable interface.

Example

The statement below creates a table function GetURL() which is implemented to return the result collection iteratively.

```
CREATE FUNCTION GetURL(url VARCHAR2) RETURN ROWSET
ITERATE USING GetURLMethods;
```

The type GetURLMethods implements the table scan routines of the ODCITable interface.

```
CREATE TYPE GetURLMethods AS OBJECT
(
```

```
        STATIC FUNCTION ODCITableStart(…)…
        MEMBER FUNCTION ODCITableFetch(…)…
        MEMBER FUNCTION ODCITableClose(…)…
);
```

Now, when the user executes a query of the form :

```
SELECT * FROM TABLE(GetURL("www.oracle.com"));
```

the table scan routines are appropriately invoked by Oracle to retrieve the rows representing the data contained in the web-site.

The differences between abstract tables and table functions lie in the dynamism of data and the set of allowed operations. The table below shows some of the key differences.

	TABLE FUNCTIONS	**ABSTRACT TABLES**
Query-time parameters	Yes	No
Create-time parameters	No	Yes
Indexes	No	Yes
Constraints and triggers	No	Yes
Inserts/deletes/updates	No	Yes
Partitioning Specification	No	Yes

Table 6: Table Functions vs. Abstract Tables

In other words, abstract tables are a more general mechanism but do not provide the ability to specify parameters at query time. Table functions are a simplified mechanism to provide user defined iterators in such cases.

Cartridge Basic Services

In order to develop and deploy full-fledged cartridges, Oracle's extensibility architecture provides a set of commonly useful routines. They represent a very useful library which not only assists in the development of data cartridges, but also facilitates inter-cartridge coordination. Typically, these basic services are exposed as OCI routines that can be invoked by a cartridge.

These cartridge service interfaces include:

- Memory Management
- Parameter Management
- Internationalization
- Error Reporting
- Context Management
- File I/O

Memory Management

The Memory Management Interfaces contain support for:

- Allocating (permanent and freeable) memory of several durations:
  - Session
  - Statement
  - User Call to Server
  - Server Call to Cartridge
- Re-allocating memory
- Allocating sub-duration memories
- Large contiguous memory allocation

and more.

Parameter Management

The parameter manager provides a set of routines to process parameters from a file or a string. Routines are provided to process the input and to obtain key and value pairs. These key and value pairs are stored in memory and can be accessed via certain routines.

The input processing routines match the contents of the file or the string against an existing grammar and compare the key names found in the input against the list of known keys that the user has registered. The behavior of the input processing routines can also be configured.

Internationalization

To support multilingual applications, national language support (NLS) **??** functionality is required by the cartridges. NLSRTL is a multi-platform and multilingual library that is currently used by Oracle ORDBMS and provides consistent NLS behavior to all Oracle products. The basic NLS services are available to cartridge developers in form of interfaces for the following functionality

- Locale information retrieval
- String manipulation in the format of multi-byte and wide-char
- Character set conversion including Unicode
- Messaging mechanism

Error Reporting

The cartridge code can return errors or raise exceptions that are handed back to the Oracle server. There are service routines to raise errors, register error messages and manipulate the error stack.

Context Management

Context management allows clients to maintain context across calls to a cartridge. The context maintained by a cartridge could be based on a duration such as SESSION, STATEMENT or CALL. The cartridge services provide a mechanism for saving and retrieving contexts.

File I/O

The OCI file I/O package is designed to make it easier to write portable code that interacts with the file system by providing a consistent view of file I/O across multiple platforms.

Case Studies

The Extensibility Services and Interfaces available in Oracle8*i* have been used by Oracle to create some commonly useful data cartridges. This section discusses the implementations of these data cartridges with a view to shedding more light on the Extensibility Architecture and its benefits.

The Oracle8*i* *inter*Media Text Data Cartridge

The Oracle8*i* *inter*Media Text Cartridge supports full-text indexing of text documents [OIMT99]. 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 Oracle8i. 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 is evaluated first. The text index is scanned and all the rows satisfying the predicate are identified. The row identifiers of all the relevant rows are written out into a temporary result table, say *results*.
2. The original query is 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 10-times improvement in performance for certain search-intensive queries, after the integration using the extensible indexing framework.

The Oracle8*i* Spatial Data Cartridge

The Oracle 8i Spatial cartridge allows users to store, spatially index and query spatial data [OSpa99]. The spatial data is modeled as an object type SDO_GEOMETRY. The coordinate values describing a geometry are stored as a collection attribute within the object type.

A spatial index can be built on a SDO_GEOMETRY column. The spatial index consists of a collection of tiles corresponding to every spatial object and is stored in an Oracle table. The spatial index is an instance of a spatial indextype which defines the routines for creating, maintaining and querying the spatial index. The spatial indextype supports an operator called Overlaps which determines which geometries in two given layers overlap with each other. A spatial querywould be of the form:

```
SELECT r.gid, p.gid  FROM roads r, parks p
WHERE Overlaps(r,p);
```

The extensible indexing framework has greatly improved the usability of the Oracle spatial cartridge. Prior to Oracle 8i, 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 this framework, the spatial index is maintained implicitly by the server just like a built in index.

Also, with the extensible indexing framework the logic of using the index to process the queries is encapsulated in the indextype routines and the end user is not burdened with any details of the index implementation. Prior to Oracle 8i the above query had to be formulated as follows:

```
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)
```

The drawback of this approach is that the querying algorithm which may be proprietary has to be exposed to the user, the entire logic has to be expressed as a single SQL statement and the user is expected to learn about the details of the storage structures for the index. In addition to vastly simplifying the queries, the Oracle 8i 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.

The Oracle8*i* Visual Information Retrieval Data Cartridge

The Oracle8*i* Visual Information Retrieval (VIR) cartridge supports content-based retrieval of images [OVIR99]. An image is modeled as the ORDImage object type. A BLOB attribute stores the raw bytes of the image. The image cartridge supports building image indexes. For purposes of building an index, each image is transformed into 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. The benefits of extensible indexing can be seen by analyzing the execution of the same image query before and in Oracle8*i*. 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 table scan, and the image comparison had to be done for every row. In Oracle8*i*, using the extensible indexing framework, the VIRSimilar operator can be 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 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 comparisons need to be performed.

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

Conclusions

Oracle8*i* provides a framework for database extensibility so that complex, content-rich types can be supported and managed natively in the database. This framework provides the infrastructure needed to allow extensions of the database server by creating domain-specific components called Data Cartridges. The major services provided by the database – Type System, Server Execution Environment, Indexing, Query Optimization, Aggregation etc. are all capable of being customized through a special set of interfaces. These Data Cartridge Interfaces help the close integration of all kinds of data into the Oracle8*i* platform.

Acknowledgments

We would like to recognize the contributions of several individuals who made the Oracle Extensibility Architecture a reality. Anil Nori, Chin Hong, Kenneth Ng, and Andy Mendelsohn helped set the direction for this technology in Oracle. There were several project leaders and developers including the authors who helped deliver the architecture; prominent among them are Jay Banerjee, Jags Srinivasan, Nipun Agarwal, Seema Sundara, Dinesh Das, S. Muralidhar and Vikas Arora.

References

[OAD99] *Oracle8i Application Developer's Guide Release 8.1.5*, Oracle Corporation, Part No. A68003-01, March 1999.

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

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

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

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

[Ora97] *Oracle8 Server Concepts Volume I and II*, Oracle Corporation, Part Number A54644-01 and A54646-01, June 1997.

[Ora99] *Oracle8i Concepts: Release 8.1.5,* Oracle Corporation, Part Number A67781-01, March 1999.

[PLSQL99] *PL/SQL User's Guide and Reference 8.1.5*, Oracle Corporation, Part Number A67842.

[OJSP99] *Oracle8i Java Stored Procedures Developer's Guide 8.1.5,* Oracle Corporation, Part No. A64686

[OJDBC99] *Oracle8i JDBC Developer's Guide and Reference 8.1.5*, Oracle Corporation, Part No. A64685

[OSQLJ99] *Oracle8i SQLJ Developer's Guide and Reference 8.1.5,* Oracle Corporation, Part No. A64684

[OEJB99] *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide 8.1.5,* Oracle Corporation, Part No. A64683