ORACLE8/8I FOR DEVELOPERS: WHAT YOU NEED TO KNOW…

John Jay King, King Training Resources

Abstract

Oracle8 and Oracle8i offer many new and enhanced features to application developers. This presentation provides an overview of major new features and how they may be used to address business and technical issues. Several examples of new features are included. In addition to understanding the potential of Oracle8/8i, current Oracle database users will learn what they need to know to begin assessing the impact of implementing Oracle8/8i in their own environments.

Introduction

Oracle8 and Oracle8i are extremely rich tools, with many new features extending the capabilities of the database in many ways. This paper focuses on those improvements and additions to Oracle8/8i likely to have the most impact on application developers. This paper assumes a working knowledge of Oracle 7, SQL, and PL/SQL (from an application developer's perspective). No prior knowledge of Oracle8 or Oracle8i is assumed.

Oracle8i includes support for Objects with all versions, but, Oracle8 supports Objects only in some installations. In this paper, Oracle8/8i's additions and enhancements have been separated into two sections: non-Object features, and Object features. The Objects Option enhances the Relational Data Base (RDBMS) providing an Object-Relational Data Base (ORDBMS). Oracle's object implementation is currently incomplete but still quite useful.

Features added or improved with Oracle8i are marked in these notes to indicate that they are Oracle8i-specific. Examples of some new features are provided during this presentation, however, you should see the Oracle8/8i SQL Reference, Oracle8/8i PL/SQL Users Guide and Reference, and other relevant Oracle manuals for complete documentation. All sample code was tested using Oracle 8.1.5 or Oracle 8.0.5. Oracle8i Release 2 (8.1.6) is due for release within days of this paper's writing, without having an opportunity to test it, only the big-ticket items will be mentioned here.

Oracle8/8i Non-Object Features

All Oracle8i installations have access to an impressive list of enhanced functionality that can be put to use immediately in creating more robust applications in addition to the new Object features. Major new or enhanced features include: Larger character data, Large Objects (BLOB, CLOB, BFILE), CUBE and ROLLUP extensions to GROUP BY, SQL enhancements, PL/SQL enhancements for large objects and directories, deferred constraints, materialized views, INSTEAD OF trigger for views, system event triggers, external procedures, PL/SQL invoker rights, advanced queuing packages (DBMS_AQ and DBMS_AQADM), RETURNING clause on UPDATE and DELETE, tablespace partitioning, reverse indexes, index-organized tables, changes to ROWID format, and several miscellaneous alterations.

Larger Character Data

Oracle8 allows larger character columns than previous versions:

Character data type	Max. in Oracle8	Max. in Oracle 7
CHAR (fixed-length strings)	2000 bytes	255 bytes
VARCHAR (variable-length strings)	4000 bytes	2000 bytes

This might prove useful in expansion of comment and text data that previously might have required LONG data or multiple columns. Unfortunately, sizes do not match PL/SQL data sizes, so longer values may potentially be truncated.

National Language Support

Oracle8 allows definition of two new character datatypes: NCHAR and NVARCHAR2. These datatypes provide storage for multi-byte character data (e.g. Unicode standard) using the National Character Set defined for the database. Many multi-byte character sets store each character in two bytes, thus, double-byte character data. Oracle8's National Language Support provides for languages requiring more flexibility. NCHAR allows fixed-length data up to 2000 bytes long

(typically 1000 double-byte characters) and NVARCHAR2 provides variable-length string data up to 4000 bytes long (typically 2000 double-byte characters). If these features interest to you, see Oracle's documentation for more detail.

Large Objects

Before Oracle8 only LONG and LONG RAW allowed larger-sized data. LONG and LONG RAW are still valid in Oracle8 and function as in Oracle 7: one LONG or LONG RAW is allowed per table with maximum size of 2GB. LONG and LONG RAW columns are stored within the table's data and might increase the time needed to scan and process rows.

Oracle8 provides four new datatypes: CLOB, BLOB, NCLOB, and BFILE. Each of the new types holds up to 4GB of data, and multiple CLOB, BLOB, and BFILE columns are allowed per table. Character Large Objects (CLOBs) and Binary Large Objects (BLOBs) are stored in the database but not necessarily within the table itself. A locator or pointer to the CLOB/BLOB is included in the table instead of the large object. NCLOB is treated the same as CLOB, but, it may contain characters in the multi-byte National Character set defined for the database. CLOB, BLOB, and NCLOB are sometimes referred to as Internal LOB data since they are stored within the database. Internal LOB data is fully covered by database backup and recovery mechanisms. BFILE data is stored outside of the database and is also known as an External LOB. BFILE contains a reference to a file stored in the host operating system. This means that BFILE data may be stored on any device accessible to the host system including CD-ROMs or hard disks. Because the BFILE references local operating system syntax, portability may be an issue. BFILE (External LOB) data is not included in Oracle8's backup and recovery mechanism. BFILE access is read-only.

CREATE TABLE and ALTER TABLE include a new LOB storage clause so that large objects may be stored separately from the main table data (partial syntax shown below).

```
CREATE TABLE xxxxx
    (col1   integer,
     col2   BLOB,
     col3   CLOB)
      STORAGE (INITIAL nn NEXT nn)
      TABLESPACE (row_data_row_ts)
      LOB (col2, col3) STORE AS
                        (TABLESPACE lob_data_ts
                          STORAGE (INITIAL nn NEXT nn)
                          CHUNK 4
                          NOCACHE LOGGING
                          INDEX (TABLESPACE lob_idx_ts
                          STORAGE (INITIAL nn NEXT nn)
     )
```

CREATE TABLE also allows specification of DISABLE STORAGE IN ROW forcing CLOB/BLOB data to be stored outside of the table, the default is ENABLE STORAGE IN ROW. ENABLE STORAGE IN ROW causes Oracle8 to store CLOB/BLOB values internally in the row until they exceed 4000 bytes (max. size of VARCHAR2) when they will be moved out of the table area. Oracle says this is probably the most efficient for smaller CLOB/BLOB values, but, if frequent table scans are caused by SELECT or UPDATE statements not using the CLOB/BLOB columns, performance may be improved by forcing separate storage. Test this, performance is likely to vary widely depending upon usage.

Oracle recommends storing LOB data and indexes associated with LOB data in a separate tablespace than the underlying table for performance purposes.

New SQL Functions for LOBs and DIRECTORY Object

Initializing a CLOB or BLOB to NULL is performed using two new built-in functions **EMPTY_CLOB()** and **EMPTY_BLOB()** a CLOB/BLOB may not be set to NULL in the same fashion as other columns. Another new function **BFILENAME()** is used to name the host file and directory in an INSERT statement. Oracle8 provides specification of a host-system directory path by means of the **CREATE DIRECTORY** command, this is useful in making file references more portable. You must have CREATE DIRECTORY or CREATE ANY DIRECTORY authority to create a directory. Once a directory exists, READ privileges should be GRANTed/REVOKEd to allow access.

```
CREATE DIRECTORY xxxx AS '/aaa/bbb/ccc';
```

Oracle8i adds the new function **TO_LOB()** that converts a LONG value to a CLOB or NCLOB, and converts a LONG RAW value to a BLOB.

New PL/SQL Package for LOBs

PL/SQL Version 8 provides a built-in package named DBMS_LOB with routines used for processing and manipulating CLOB, BLOB, NCLOB, and BFILE data. Listed below are the names of PL/SQL routines included in DBMS_LOB, separated according to function.

To read or examine values in a LOB:

DBMS_LOB.GETLENGTH	Returns length of LOB
DBMS_LOB.INSTR	Finds position of specified string in LOB
DBMS_LOB.READ	Get data from LOB starting at specified offset
DBMS_LOB.SUBSTR	Returns specified part of LOB

To alter values in a CLOB, BLOB, or NCLOB:

DBMS_LOB.APPEND	Adds LOB to end of another LOB
DBMS_LOB.COPY	Copy all or part of one LOB to another
DBMS_LOB.ERASE	Delete all or part of a LOB
DBMS_LOB.LOADFROMFILE	Loads BFILE data into an internal LOB
DBMS_LOB.TRIM	Reduces LOB to specified length
DBMS_LOB.WRITE	Write data to LOB beginning at specified offset

To read values in a BFILE:

DBMS_LOB.FILECLOSE	Close the associated file
DBMS_LOB.FILECLOSEALL	Close all files opened with FILEOPEN
DBMS_LOB.FILEEXISTS	Check for presence of file on server
DBMS_LOB.FILEGETNAME	Gets directory alias and file name
DBMS_LOB.FILEISOPEN	Check to see if file was opened
DBMS_LOB.FILEOPEN	Open file

DBMS_LOB errors might generate four named PL/SQL exceptions: INVALID_ARGVAL, ACCESS_ERROR, INSERT, NO_DATA_FOUND, and VALUE_ERROR.

Deferred Constraints

Sometimes honoring Referential Integrity constraints makes logic more difficult that it needs to be. Oracle8 allows constraints to be marked as DEFERRABLE when the constraint is created/altered. When constraints are deferred, DML statements (UPDATE, INSERT, DELETE) may perform activities that might be disallowed by constraints. Constraints will then be enforced when a COMMIT occurs. The initial setting for the DEFERRABLE constraint might specify DEFERRABLE INITIALLY IMMEDIATE causing constraints to be checked at the end of a DML statement unless the transaction is modified to allow DEFERRED constraints (the default). The initial setting for the DEFERRABLE constraint might also be DEFERRABLE INTIALLY DEFERRED causing constraint checking to wait until the end of the transaction (COMMIT). The ALTER SESSION command may be used to defer constraints until a COMMIT point, but, it only impacts constraints marked as DEFERRABLE.

Example CREATE TABLE syntax for DEFERRABLE:

```
CREATE TABLE …
    CONSTRAINT …
           DEFERRABLE  INITIALLY IMMEDIATE   or
           DEFERRABLE INITIALLY DEFERRED     or
           NOT DEFERRABLE
        …
```

Example use of deferred constraints:

```
    ALTER SESSION SET CONSTRAINTS=DEFERRED;
    UPDATE emp
        SET deptno = 13
```

```
    WHERE deptno = 10;
UPDATE dept
    SET deptno = 13
    WHERE deptno = 10;
COMMIT;
```

Read-Only Views and INSTEAD OF Trigger for Views

To make a view read-only, add the WITH READ ONLY clause.

```
CREATE OR REPLACE VIEW xxx
    AS SELECT …
        WITH READ ONLY;
```

UPDATEs and INSERTs into views based upon a JOIN has been possible in Oracle for a while. However, the rules for making a view updateable and insertable are sometimes cumbersome to deal with. Oracle8 provides the option of defining INSTEAD OF triggers for the view. These triggers fire instead of Oracle8 attempting to perform DML (INSERT/UPDATE/DELETE) against the underlying tables. The values of the new/updated row are available to the trigger using the correlation name :NEW.

```
CREATE OR REPLACE TRIGGER xxx
    INSTEAD OF INSERT ON myview       or INSTEAD OF UPDATE          or INSTEAD OF DELETE
    DECLARE
        …
    BEGIN
        …
/* Code to manipulate necessary tables using :NEW values */
        …
    END;
```

External Procedures

PL/SQL Version 8 allows execution of procedures written in host-specific programming languages as External Procedures. This is accomplished in a multiple-step process: Program is written/purchased and installed as a dll-style executable in the host environment, Net8 (formerly SQL*Net) listener is modified to watch for the external process, a LIBRARY is created in Oracle8 (use CREATE LIBRARY) identifying the path to the shared library where the executable module resides, and a PL/SQL procedure (known as a Wrapper Procedure) is created to act as an interface between PL/SQL blocks in Oracle8 and the host program

Finally, any PL/SQL procedure can execute the Wrapper Procedure causing the host program to execute

```
… /* Create LIBRARY pointing to place where executable may be found */
    CREATE LIBRARY xxx AS '/aaa/bbb/ccc.lib';
… /* Create Wrapper procedure */
    CREATE OR REPLACE PROCEDURE yyy
        AS EXTERNAL
        LIBRARY xxx
        NAME zzzz
        LANGUAGE C     (not required, C only language supported by 8.0.3)
        CALLING STANDARD C            or             CALLING STANDARD PASCAL
        PARAMETERS (parm1             STRING,
                    parm2             INT);
        -- Parameter types documented in PL/SQL User's Guide and Reference
… /* Anonymous PL/SQL block using the external procedure */
    BEGIN
        yyy('instring','inval);
    END;
```

Advanced Queuing Packages (DBMS_AQ and DBMS_AQADM)

Communications between sessions has been handled using the DBMS_PIPE and DBMS_ALERT packages in many applications. Oracle8 provides a new table-based mechanism, Advanced Queuing, designed specifically to provide reliable communications between Oracle8 sessions. Each session is allowed to place an ENQUEUE into an enqueue table that is available to other sessions. Since the ENQUEUE information is stored in a table, it is protected like all other Oracle data. Many features are available through the use of two built-in PL/SQL packages, DBMS_AQ and DBMS_AQADM. DBMS_AQADM is used for creating queue tables and controlling who is allowed to use them. DBMS_AQ.ENQUEUE and DBMS_AQ.DEQUEUE are then used by applications to add/remove entries from the queue table.

Miscellaneous SQL Enhancements

Oracle8 includes a variety of SQL modifications, mostly to support new features. Oracle V6 compatibility mode has been eliminated, so, SQL and programs expecting the old behavior must be changed. Some other enhancements have been added to support Objects.

Oracle8i allows subqueries just about anywhere within the SQL statement.

UPDATE, and DELETE support a new RETURNING clause providing the ability to get values after the DML statement has completed. This behaves like the INTO clause when the statements are embedded in host programs or PL/SQL.

```
UPDATE emp
    SET SAL = SAL * 1.1
    WHERE JOB = 'CLERK'
    RETURNING SAL INTO :NEWSAL;
```

All DML statements (SELECT, INSERT, UPDATE, and DELETE) may reference the specific PARTITION that rows to be operated on resides in. This may be a good performance move, but, builds a potential maintenance time-bomb into the application. If indexes have been partitioned and index columns are referenced in the WHERE clause, this may be unnecessary.

```
DELETE FROM xxx PARTITION (yyy) WHERE … ;
```

Oracle8i Release 2 (8.1.6) uses a new default date format with a four-digit year (YYYY or RRRR).

CUBE and ROLLUP Extensions to GROUP BY (Oracle8i)

The increased emphasis on data-mining activities includes a need for super-aggregate capabilities. CUBE and ROLLUP extend the ability of GROUP BY to include some of these features. ROLLUP builds subtotal aggregates at any level requested, including grand total. CUBE extends ROLLUP to calculate all possible combinations of subtotals for a specific GROUP BY. Cross-tabulation reports are created easily using CUBE. This statement shows the impact of ROLLUP:

```
SQL> col deptid format a6
SQL> run
  1  select nvl(to_char(deptno),'Grand') deptid,
  2         nvl(job,'Total') job,
  3         sum(sal) as sal
  4  from emp
  5* group by rollup (deptno,job)
DEPTID JOB              SAL
------ --------- ----------
10     CLERK           1300
10     MANAGER         2450
10     PRESIDENT       5000
10     Total           8750
20     ANALYST         6000
20     CLERK           1900
20     MANAGER         2975
20     Total          10875
30     CLERK            950
30     MANAGER         2850
30     SALESMAN        5600
30     Total           9400
Grand  Total          29025
```

Note that ROLLUP creates subtotals for each level of subtotal, and a grand total. The subtotal and grand total lines substitute NULL for columns (see use of NVL above). To improve dealing with these NULLs, Oracle provides the GROUPING function. GROUPING returns a value of 1 if a row is a subtotal created by ROLLUP or CUBE, and a 0 otherwise.

```
  1  select decode(grouping(deptno),1,'All Departments',deptno) deptno
  2        ,decode(grouping(job),1,'All Jobs',job) job
  3        ,sum(sal) as sal
  4  from emp
  5* group by rollup (deptno,job)
DEPTNO                                JOB              SAL
------------------------------------- --------- ----------
10                                    CLERK           1300
10                                    MANAGER         2450
10                                    PRESIDENT       5000
10                                    All Jobs        8750
20                                    ANALYST         6000
```

```
20                                  CLERK          1900
20                                  MANAGER        2975
20                                  All Jobs      10875
30                                  CLERK           950
30                                  MANAGER        2850
30                                  SALESMAN       5600
30                                  All Jobs       9400
All Departments                     All Jobs      29025
```

CUBE automatically creates all possible combinations from the available subtotals:

```
SQL> run
  1  select decode(grouping(deptno),1,'All Departments',deptno) deptno
  2        ,decode(grouping(job),1,'All Jobs',job) job
  3        ,sum(sal) as sal
  4  from emp
  5* group by cube (deptno,job)
DEPTNO                              JOB              SAL
----------------------------------- --------- ----------
10                                  CLERK           1300
10                                  MANAGER         2450
10                                  PRESIDENT       5000
10                                  All Jobs        8750
20                                  ANALYST         6000
20                                  CLERK           1900
20                                  MANAGER         2975
20                                  All Jobs       10875
30                                  CLERK            950
30                                  MANAGER         2850
30                                  SALESMAN        5600
30                                  All Jobs        9400
All Departments                     ANALYST         6000
All Departments                     CLERK           4150
All Departments                     MANAGER         8275
All Departments                     PRESIDENT       5000
All Departments                     SALESMAN        5600
All Departments                     All Jobs       29025
```

Oracle8i Release 2 (8.1.6) adds “Analytic” functions allowing ranking, moving aggregates, period comparisons, ratio of total, cumulative aggregates, and more.

Index using Function Call or Expression (Oracle8i)

Normally, developers attempt to use index columns wherever possible to speed access to large database tables. Until now, this has often been hampered by limitations of index design. Until Oracle8i index columns always represented the actual value of the column in the database and an SQL WHERE clause needed to specify the unadulterated column value or the index would be ignored. Now, an index can represent a column value after some function or expression has been applied. As long as an SQL WHERE clause references a column value after a function or expression exactly as specified in the index creation, an index may be used by the system. Given the following create index lines:

```
CREATE INDEX … ON EMP (UPPER(ENAME)) …
CREATE INDEX … ON EMP (NVL(SAL,0)+NVL(COMM,0)) …
```

The following two WHERE clause lines may be indexed by the optimizer (requires cost-based optimization).

```
SELECT … WHERE UPPER(ENAME) = UPPER(:hostvar) …
SELECT … WHERE NVL(SAL,0)+NVL(COMM,0) > 1000 …
```

DDL and Database Event Triggers (Oracle8i)

We have used database triggers happily since they first appeared with Oracle 7. Oracle8i now calls the traditional INSERT, UPDATE, and DELETE triggers "DML" triggers to differentiate them from the new "DDL" and "Database event" triggers. The new triggers offer the ability to control the environment more completely than before.

DDL triggers fire due to CREATE, ALTER, or DROP statements:

BEFORE CREATE or AFTER CREATE	When a catalog object is created
BEFORE ALTER or AFTER ALTER	When a catalog object definition is changed

BEFORE DROP or AFTER DROP	When a catalog object is dropped

Database event triggers fire when specific system-level events occur:

LOGON	After successful LOGON of new user
LOGOFF	At beginning of LOGOFF process
SERVERERROR	When any server error, or a specific error number occurs
STARTUP	When database is opened
SHUTDOWN	At beginning of instance shutdown

Materialized Views (Oracle8i)

Oracle's SNAPSHOT is a query result table that is created periodically to facilitate distribution or replication of data. the materialized view feature of Oracle8i uses similar technology to allow a view's results to be stored as materialized in the database for use by subsequent SQL statements. The view materialization is refreshed periodically based upon time criteria specified when the view is created or upon demand. View data is "old" until the view is refreshed. This is an ideal mechanism for improving the performance of frequent requests for aggregate data. Oracle8i Release 2 (8.1.6) allows ORDER BY in the Materialized View definition.

```
create materialized view dept_summary
  refresh start with sysdate next sysdate + 1
as
  select dname,count(*) nbr_emps,sum(nvl(sal,0)) tot_sal
        from emp,dept
        where emp.deptno(+) = dept.deptno
        group by dname
```

PL/SQL Invoker Rights (Oracle8i)

Stored procedures, functions, and packages have been improving the quality of applications for some time now. By default, when stored PL/SQL is executed, the code executes under the security domain of the userid used to compile the stored PL/SQL. The means that users not normally able to perform specific database actions directly, can gain indirect permission by virtue of executing stored PL/SQL. This is often a useful feature. However, occasionally, it might be useful to require the user executing stored PL/SQL to have the security authorization to perform all actions contained within the code. Oracle8i provides a new clause on the CREATE PROCEDURE, CREATE FUNCTION, and CREATE PACKAGE statements allowing control over the security domain used at execution time. The default, if nothing is specified, is to have the PL/SQL execute under the security domain of the userid that compiles the code. This provides complete upward compatibility for all code written before Oracle8i. The two controlling options are coded as follows:

create procedure xxx (parameter list) AUTHID DEFINER as … pl/sql block …

create procedure yyy (parameter list) AUTHID CURRENT_USER as … pl/sql block …

AUTHID DEFINER is the default and works the way all previous stored PL/SQL has using the security domain of the user that compiles the code. AUTHID CURRENT_USER indicates that the current user's id must be capable of all actions being taken by the stored PL/SQL.

Autonomous Transactions (Oracle8i)

Occasionally it is useful for a body of work being performed to COMMIT/ROLLBACK without regard to other code in use. Prior to Oracle8i this was partially possible via SAVEPOINT for limited ROLLBACK control, but, no COMMIT support was available. Autonomous transactions allow a COMMIT/ROLLBACK transaction sequence within a code block that is not connected to the COMMIT/ROLLBACK in the outer transaction. This is accomplished by placing the following line in **the declarative section** of any anonymous PL/SQL block, Procedure, or Function.

```
declare
    …
    pragma autonomous_transaction;
    …
begin
    …
```

```
function xxx (parm list) return vartype is
                …
   pragma autonomous_transaction;
   …
begin
   …
```

When the BEGIN block following this declaration is executed, the current transaction is temporarily stopped and a new transaction begins. Any COMMIT/ROLLBACK sequence (including SAVEPOINTs) that occurs in the block is treated

as a separate transaction. Upon reaching the END; of the block, the original transaction is reinstated and is unaffected by any COMMIT/ROLLBACK that happened during the autonomous transaction. The autonomous transaction must COMMIT or ROLLBACK before ending. Autonomous transactions may be nested. Within a package, each procedure or function that will have an autonomous transaction must specify the **pragma autonomous transaction clause**, there is no provision for setting multiple functions or procedures to be autonomous.

Temporary Tables (Oracle8i)

Temporary Tables were added due to popular demand to provide a table that is visible to a single transaction or session. All DML and TRUNCATE TABLE may be used with Temporary Tables. Indexes and synonyms may be created for them too. Temporary Table definitions may be shared by many transactions or sessions, but, each transaction or session gets its own copy of the data. All data is deleted when the transaction or session ends. The CREATE TABLE syntax indicates when something is a temporary table and whether it is session or transaction specific.

```
CREATE GLOBAL TEMPORARY TABLE xxx ON COMMIT PRESERVE ROWS …          *** Session specific
CREATE GLOBAL TEMPORARY TABLE yyy ON COMMIT DELETE ROWS …            *** Transaction specific
```

Transactions generate UNDO information for Temporary Tables, but, not REDO information. Table and index data disappears at the end of the transaction (ON COMMIT DELETE) or at the end of the session (ON COMMIT PRESERVE), but, the table description and any index definitions remain for later use.

Java (Oracle8i)

Many papers are devoted to the topic of Java and Oracle8i. Rather than add to the large body of work currently available, just the significant highlights are noted here. Oracle8i includes a Java Virtual Machine specifically engineered by Oracle to provide multi-threaded support of Java applications instead of having separate JVMs for each bit of Java. This means greatly improved performance for Java applications running with database code. Oracle also supports the creation of stored procedures using Java in addition to those created with PL/SQL. Java support for programming includes: Java stored procedures, Enterprise Java Bean 1.0 support, and support for CORBA 2.0 standards. Java support for SQL includes: JDBC and SQLJ. SQLJ statements are translated by an SQLJ Preprocessor before Java code is submitted to JDBC. Direct JDBC support is more complex, but, yields more control.

Oracle8i Release 2 (8.1.6) dramatically improves Java support including: Java 2 (JDK 1.2) support, JDBC 2.0, multi-byte character support, debugging support, performance improvements, and support for Advanced Queueing. The new release also supports an XML parser implemented with Java.

Tablespace Partitioning (DBA issue you might be interested in)

If you have very large tables, you want to make sure your DBA's are aware of this feature. Beginning with Oracle8 it is possible to associate one table's data with a series of Tablespace Partitions so that a large table's data may be spread out easily across multiple devices. Each Tablespace Partition is represented by one-or-more files in the underlying host environment. This greatly aids in performance, especially if the Parallel Processing features of Oracle8 are being used. By using Partitioned Indexes data from the underlying table may be placed into specific Tablespace Partitions by key range. The optimizer is partition aware and will generate code to search only the relevant partitions. Once created, Tablespace Partitions are largely invisible to application developers unless they specifically choose to reference a particular partition with SQL. Specific partition references may be an interesting feature in terms of speed, but, may prove to add a new layer of maintenance issues.

Reverse Key Indexes (DBA issue you might be interested in)

If keys and usage are heavily clustered in a table (e.g. key is generated as next sequential number, and most activity occurs on recent entries), then, Reverse Key Indexes might speed things up. A Reverse Key Index is simply a standard index with the key values stored in reversed form (e.g. '1234' becomes '4321', '1235' becomes '5321'). Actual table data is not changed. This causes keys to be more evenly distributed reducing the amount of work that must take place to "balance" the index. This is especially useful in Parallel Processing environments but is also useful any time key ranges become too dense and cause extra processing due to the key density.

Index-Organized Tables (DBA issue you might be interested in)

CREATE TABLE now has a new clause ORGANIZATION INDEX that causes the table's data to be incorporated into the B-tree index representing the table's Primary Key. This means that the table's data is always available in sequence by Primary Key and many sorts may be avoided by the optimizer. Oracle8i adds the ability to create secondary indexes for Index-Organized tables (not allowed in Oracle8.0). This feature is especially useful for "lookup" tables often used with drop-down lists in GUI interfaces, the data is already be in the desired sequence so access may be speeded.

Change to ROWID (DBA issue you might be interested in)

The actual format of the ROWID has changed in Oracle8. For most developers this is a non-issue since the external form of a ROWID is still an 18-character external representation. Applications that simply store ROWID and use it later will be unaffected by this change. Unfortunately, some developers (oh, what a tangled web we weave…) have applications that for one reason or another parse the ROWID and make decisions based upon the contents. For that brave group, here is a comparison of the old ROWID and the new ROWID. For more specifics, see the Oracle documentation.

Oracle 7 ROWID Internal format 6-bytes

External format 18-bytes: BBBBBBBB.RRRR.FFFF

B = Block number, R = Row Number, F = File Number

Oracle8/8i ROWID Internal format 10-bytes

External format 18-bytes: OOOOOOFFFBBBBBBSSS

O = Database Object number, F = Relative File Number,

B = Block number, S = Slot/Row Number

Direct-path loading from C program (Oracle8i)

OCI programs may use direct-path loading in the same way that SQL*Loader does to speed the movement of data from non-Oracle sources into the database. Like SQL*Loader the speed is gained by avoiding any constraints or triggers, and by lack of support for user-defined datatypes. LOB must be loaded after all scalar columns, and LONG data must be loaded last.

ALTER TABLE DROP COLUMN (Oracle8i)

The DROP COLUMN clause has been added to provide a capability requested by Oracle users for quite a long time. PL/SQL's dependency checks will mark stored PL/SQL invalid that uses the column, however, columns referenced in user applications must be checked and modified manually. Another option has been added to ALTER TABLE called SET UNUSED, this causes the values of a column to become inaccessible without taking the time to actually delete the column. However, the unusable column remains as part of the table until DROP COLUMN is executed.

Oracle8/8i Object Features

Oracle8/8i's Objects Option adds an exciting new dimension to Oracle application development. While still awaiting full object implementation, Oracle8 added the ability to create User-defined data types, Nested Tables, Varrays, REFs, Object Views, Encapsulation of attributes and methods within an object type. Oracle8i improved the original features. Both SQL and PL/SQL provide new features used to deal with the object additions and enhancements.

User-Defined Data Types

Occasionally the datatypes provided by relational database are not sufficient. For instance, relational theory allows only atomic datatypes and does not provide for groups of attributes that might be used sometimes collectively and other times as individual attributes. The classic example of this is an address. A typical address might include House Number, Street, City, State, Mail Code, and maybe Country. As developers in the SQL world, we tend to keep the separate attributes distinct to provide robustness in our data design. Oracle8 allows the best of both worlds:

```
create or replace type address as object
```

```
( house_number number(6),
  street1 varchar2(30),
  street2 varchar2(30),
  city    varchar2(20),
  state   varchar2(20),
  mailcode varchar2(15),
  country varchar2(20)
);
create table employee
( empid   number(4) not null primary key,
  lastname varchar2(30),
  firstname varchar2(30),
  home_address address);
create table purchase_order
( po_number  number(6) not null primary key,
  customer_name varchar2(30) not null,
  shipping_address address,
  billing_address address);
```

Encapsulation of Attributes and Methods

User-defined objects may include both attributes (data items) and methods (procedures/functions) in an object type. This is not considered true encapsulation since member attributes may be modified by either method or by non-method SQL.

For example:

```
create or replace type cust_order_type as object
    (po_number number(6),
     customer  varchar2(30),
     billing_address address,
     shipping_address address,
     order_date date,
     member function days_old return number,
      pragma restrict_references (days_old,wnds,wnps)  );
create or replace type body cust_order_type
     as member function days_old
      return number
    is
    begin
      return sysdate - order_date;
    end days_old;
end;
/
```

Nested Tables

Nested tables are an interesting offshoot allowing us to create tables of data where the contents of another table are included within a row of the table. For instance, suppose your design had a table that listed Purchase Orders and another table with Purchase Order Lines. It is unlikely that the two tables would be used independently and a join would be required for processing. Nested tables would allow all of the Purchase Order Lines for a specific Purchase Order to be obtained by means of the row, all Purchase Order Lines pertinent to a Purchase Order would simply be there, attached to the Purchase Order. Nested tables requires some interesting new syntax. The examples below use the Oracle sample tables DEPT and EMP as a basis so that you can experiment with them on your own system.

```
create or replace type deptemp
    as object
    (EMPNO        NUMBER(4),
     ENAME        VARCHAR2(10),
     JOB          VARCHAR2(9),
     MGR          NUMBER(4),
     HIREDATE     DATE,
     SAL          NUMBER(7,2),
     COMM         NUMBER(7,2) );
/
create or replace type deptemps as table of deptemp;
/
create table department
    (deptno number(2) not null,
     dname  varchar2(15),
     loc    varchar2(20),
     employees deptemps)
     nested table employees store as emps;
/
```

```
insert into department
     (deptno,dname,loc,employees)
     select deptno,dname,loc,
        cast(multiset(select empno,ename,job,mgr,hiredate,sal,comm
                    from emp where deptno = dept.deptno
                  )
             as deptemps
        )
    from dept;
/
select deptno,dname,loc
    from department
    where deptno = 20;
 select empno,ename
    from the (select employees from department where deptno = 20);
```

Varrays

Varying arrays (Varrays) are similar to nested tables in definition and use. This is useful when a set of data (probably defined by a user object) may occur only a specified number of times. The VARRAY option specifies the maximum number of sets allowed in the varying array.

```
    create or replace type deptemp
        as object
        (EMPNO        NUMBER(4),
         ENAME        VARCHAR2(10),
         JOB          VARCHAR2(9),
         MGR          NUMBER(4),
         HIREDATE     DATE,
         SAL          NUMBER(7,2),
         COMM         NUMBER(7,2)  );
    /
    create or replace type deptemps as varying array (10) of deptemp;
    /
    create table department
        (deptno number(2) not null,
         dname  varchar2(15),
         loc    varchar2(20),
         employees deptemps);
```

Object Tables

Tables may be created representing an object type. This is a simplistic example, but, should get the point across.

```
create or replace type emp_type
as object
(EMPNO        NUMBER(4),
 ENAME        VARCHAR2(10),
 JOB          VARCHAR2(9),
 MGR          NUMBER(4),
 HIREDATE     DATE,
 SAL          NUMBER(7,2),
 COMM         NUMBER(7,2)  );
/
create table my_emps of emp_type
    (empno primary key not null, hiredate not null);
insert into my_emps
    values (emp_type(1234,'WU','NET HERO',NULL,sysdate,60000,20000));
```

Object Views

Views may be based upon object types. The only change is the WITH OBJECT OID (col) clause that contains the attribute (or list of attributes) used to uniquely identify a row in the view. Object Views also allow the use of object technology with existing tables. There are four steps to creating an Object View using EMP data as a nested table:

Define object identically to relational table

Define object view using relational table

Define new object type and object table

Define view using nested table syntax

Define the object first:

```
create or replace type jempobj as object
  (EMPNO           NUMBER(4),
   ENAME           VARCHAR2(10),
   JOB             VARCHAR2(9),
   MGR             NUMBER(4),
   HIREDATE        DATE,
   SAL             NUMBER(7,2),
   COMM            NUMBER(7,2),
   DEPTNO          NUMBER(2) )
/
create or replace type jempobj as object
  (EMPNO           NUMBER(4),
   ENAME           VARCHAR2(10),
   JOB             VARCHAR2(9),
   MGR             NUMBER(4),
   HIREDATE        DATE,
   SAL             NUMBER(7,2),
   COMM            NUMBER(7,2),
   DEPTNO          NUMBER(2) )
/
```

Next, Define an Object View and Object Table

```
create or replace view jemp_obj_view of jempobj
  with object oid (empno) as
  select   empno,
           ename,
           job,
           mgr,
           hiredate,
           sal,
           comm,
           deptno
    from emp
/
create type jemp_n_table as table of jempobj;
/
```

Then, define a View using Nested Table Syntax

```
create or replace view jemp_o_view (deptno,dname,emptab) as
 select dept.deptno,dept.dname
        ,cast(multiset
           (select  emp.empno
                   ,emp.ename
                   ,emp.job
                   ,emp.mgr
                   ,emp.hiredate
                   ,emp.sal
                   ,emp.comm
                   ,emp.deptno
              from emp
              where emp.deptno = dept.deptno)
         as jemp_n_table)
   from dept
/
```

Finally, use the Relational Table via the Object View

```
select ename from
   the  (select emptab from jemp_o_view where deptno = 20)
/
```

REF and VALUE

REF() is a built-in function used to obtain the address of an object in the database. This may be useful inside PL/SQL code blocks where the REF may be captured with an INTO clause and reused later.

```
select ref(r) from my_emps r;

REF(R)
--------------------------------------------------------------------------------
0000280209AC710BDDB43411D1ABE400A0C9096AC3AC710BDCB43411D1ABE400A0C9096AC3004035080000
```

VALUE() is a built-in function that returns the object rather than the attributes that make up the object.

```
select * from my_emps;

EMPNO      ENAME      JOB           MGR           HIREDATE SAL   COMM
---------  ---------- ---------     ---------     --------- ---------   ---------
1234       WU         NET HERO                    05-MAR-98      60000   20000

select value(r) from my_emps r;

VALUE(R)(EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM)
-----------------------------------------------------------------------
EMP_TYPE(1234, 'WU', 'NET HERO', NULL, '05-MAR-98', 60000, 20000)
```

Other SQL and PL/SQL

SQL and PL/SQL both provide additional new features used to deal with the object additions and enhancements. These include a SELF pointer and DEREF() function among the others.

Conclusion

This presentation attempted to illustrate the major features of Oracle8 and Oracle8i impacting application developers who need to create new systems or change existing systems. Whether or not your organization takes advantage of the object-oriented or Java-oriented portions, there are many new features ready and waiting to make your applications more useful. This has not been a comprehensive look at all of the new and improved features of Oracle8 and Oracle8i, many features not directly applicable to application development (such as new security features) have been omitted. In addition, some topics have been simply exposed rather than fully explored in the interest of fitting into the time allowed. Oracle8/8i is a rich development environment, improving on the already full-functioned capabilities of the Oracle 7 database product. As application developers these improved tools will allow us to better serve the business needs of our customers.

About the Author

John King is a Partner in King Training Resources, a firm providing instructor-led training since 1988 across the United States and Internationally. John has worked with Oracle products since Version 4 and has been providing training to application developers since Oracle Version 5. He has presented papers at various industry events including: IOUG-A Live!, UKOUG Conference, EOUG Conference, ECO/SEOUC, RMOUG Training Days, and the ODTUG conference.

John Jay King
King Training Resources
6341 South Williams Street
Littleton, CO 80121-2627 U.S.A.
Phone: 1.800.252.0652 1.303.798.5727
Fax: 1.303.730.8542
Email: john@kingtraining.com Website: www.kingtraining.com

If you have any questions about our training services, this paper, or comments, please contact us. Copies of this paper may be downloaded from the King Training Resources website (www.kingtraining.com).

Bibliography

Oracle8i SQL Reference, Oracle Corporation

Oracle8i Concepts, Oracle Corporation

Oracle8i PL/SQL User's Guide and Reference, Oracle Corporation

Oracle8i Application Developer's Guide (Fundamentals, Advanced Queuing, Large Objects), Oracle Corporation

Oracle8i Administrator's Guide, Oracle Corporation

Oracle8i Supplied Packages Reference