1

Design Basics

This chapter explains several important design goals for database developers.

This chapter contains the following topics:

See Also:

- Database Development Fundamentals for high-level database concepts
- *Oracle Database 2 Day Developer's Guide* for more information about Oracle Database concepts and techniques

1.1 Design for Performance

The key to database and application performance is design, not tuning. While tuning is quite valuable, it cannot make up for poor design. Your design must start with an efficient data model, well-defined performance goals and metrics, and a sensible benchmarking strategy. Otherwise, you will encounter problems during implementation, when no amount of tuning will produce the results that you could have obtained with good design. You might have to redesign the system later, despite having tuned the original poor design.

See Also:

- Performance and Scalability
- *Oracle Database Performance Tuning Guide*
- *Oracle Database SQL Tuning Guide*

1.2 Design for Scalability

Scalability is the ability of a system to perform well as its load increases. Load is a combination of number of data volumes, number of users, and other relevant factors. To design for scalability, you must use an effective benchmarking strategy, appropriate application development techniques (such as bind variables), and appropriate Oracle Database architectural features like shared server connections, clustering, partitioning, and parallel operations.

See Also:

- Performance and Scalability
- *Database 2 Day Developer's Guide*

1.3 Design for Extensibility

Extensibility is the ease with which a database or database application accommodates future growth. The more extensible the database or application, the easier it is to add or change functionality with minimal impact on existing functionality.

Note:

Extensibility differs from **forward compatibility**, the ability of an application to accept data from a future version of itself and use only the data that it was designed to accept.

For example, suppose that an early version of an application processes only text and a later version of the same application processes both text and graphics. If the early version can accept both text and graphics, and ignore the graphics and process the text, then it is forward-compatible. If the early version can be upgraded to process both text and graphics, then it is extensible. The easier it is to upgrade the application, the more extensible it is.

To maximize extensibility, you must design it into your database and applications by including mechanisms that allow enhancement without major changes to infrastructure. Early versions of the database or application might not use these mechanisms, and perhaps no version will ever use all of them, but they are essential to easy maintenance and avoiding early obsolescence.

Topics:

- Data Cartridges
- External Procedures
- User-Defined Functions and Aggregate Functions
- Object-Relational Features

1.3.1 Data Cartridges

Data cartridges extend the capabilities of the Oracle Database server by taking advantage of Oracle Extensibility Architecture framework. This framework lets you capture business logic and processes associated with specialized or domain-specific data in user-defined data types. Data cartridges that provide new behavior without using additional attributes can do so with packages rather than user-defined data types. With either user-defined types or packages, you determine how the server interprets, stores, retrieves, and indexes the application data. Data cartridges package this functionality, creating software components that plug into a server and extend its capabilities into a new domain, making the database itself extensible.

You can customize the indexing and query optimization mechanisms of an extensible database management system and provide specialized services or more efficient processing for user-defined business objects and rich types. When you register your implementations with the server through extensibility interfaces, you direct the server to implement your customized processing instructions instead of its own default processes.

Related Topics

- *Oracle Database Data Cartridge Developer's Guide*

1.3.2 External Procedures

External procedures are highly extensible because you can enhance their functionality without affecting their invokers. The reason is that the call specification of an external procedure, which has all the information needed to invoke it, is separate from the body of the procedure, which has the implementation details. If you change only the body, and not the specification, then invokers are unaffected.

Related Topics

- Developing Applications with Multiple Programming Languages

1.3.3 User-Defined Functions and Aggregate Functions

User-defined PL/SQL functions that can appear in SQL statements or expressions can extend the functionality of SQL.

User-defined aggregate functions are part of the Oracle Extensibility Architecture framework.

See Also:

- Invoking Stored PL/SQL Functions from SQL Statements for information about invoking user-defined PL/SQL functions in SQL statements and expressions
- *Oracle Database Data Cartridge Developer's Guide* for information about user-defined aggregate functions

1.3.4 Object-Relational Features

The object relational features of Oracle Database are the user-defined Abstract Data Types (ADTs). ADTs are highly extensible because you can enhance their functionality without affecting their invokers. The reason is that the call specification of an external procedure, which has all the information needed to invoke it, is separate from the body of the procedure, which has the implementation details. If you change only the body, and not the specification, then invokers are unaffected.

See Also:

- *Oracle Database PL/SQL Language Reference* for information about the `CREATE TYPE` statement
- *Oracle Database PL/SQL Language Reference* for information about the `CREATE TYPE BODY` statement
- *Oracle Database Object-Relational Developer's Guide* for more information about object relational features

1.4 Design for Security

Database security involves a wide range of potential activities, including:

- Designing and implementing security policies to protect the data of an organization, users, and applications from accidental, inappropriate, or unauthorized actions
- Creating and enforcing policies and practices of auditing and accountability for inappropriate or unauthorized actions
- Creating, maintaining, and terminating user accounts, passwords, roles, and privileges
- Developing applications that provide desired services securely in a variety of computational models, leveraging database and directory services to maximize both efficiency and ease of use

See Also:

- *Oracle Database Security Guide* for more information about security.
- Security for information about considerations and techniques for providing security.

1.5 Design for Availability

Availability is the degree to which an application, service, or function is accessible on demand. A system designed for high availability provides uninterrupted computing services during essential time periods, during most hours of the day throughout the year, with minimal downtime for operations such as upgrading the system's hardware or software. The main characteristics of a highly available system are:

- Reliability
- Recoverability
- Timely error detection
- Continuous operation

See Also:

- High Availability for information about important considerations and techniques for providing high availability.

1.6 Design for Portability

While PL/SQL is not designed for portability between Oracle Database and third-party databases, it is highly portable across operating systems and languages. Most programming languages can invoke PL/SQL, and PL/SQL is implemented consistently on every platform that supports Oracle Database, including Macintosh, Linux, and Windows. If you develop PL/SQL applications on one platform, you can be highly confident that they will work consistently on all other platforms.

PL/SQL stored procedures provide some application portability across multiple databases. Although using stored procedures written in the language of a given vendor may seem to tie you to that vendor to some extent, stored procedures make the application's visual component (user interface) and application logic portable. The data logic is encoded optimally for the database on which the application runs. Because the data logic is hidden in stored procedures, you can use the vendor's extensions and features to optimize the data layer.

When developed and deployed on a database, the application can stay deployed on that database forever. If the application is moved to another database, the visual component and application logic can move independently of the data logic in the stored procedures, which simplifies the move. (Reworking the application in combination with the move complicates the move.)

See Also:

Oracle Database PL/SQL Language Reference for conceptual, usage, and reference information about PL/SQL

1.7 Design for Diagnosability

Oracle Database includes a fault diagnosability infrastructure for preventing, detecting, diagnosing, and resolving database problems. Problems include critical errors such as code bugs, metadata corruption, and customer data corruption. The goals of the diagnosability infrastructure are to detect problems proactively, limit damage and interruptions after a problem is detected, reduce the time required to diagnose and resolve problems, and simplify any possible interaction with Oracle Support.

Automatic Diagnostic Repository (ADR) is a file-based repository that stores database diagnostic data such as trace files, the alert log, and Health Monitor reports. ADR is located outside the database, which enables Oracle Database to access and manage ADR when the physical database is unavailable.

See Also:

- *Oracle Database Concepts* for an overview of diagnostic files
- *Oracle Database Administrator's Guide* for detailed information about the Oracle Database fault diagnosability infrastructure.

1.8 Design for Special Environments

This topic introduces designing for the following special database and application environments:

- Data Warehousing
- Online Transaction Processing (OLTP)

1.8.1 Data Warehousing

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data, but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources. This strategy helps the organization maintain historical records and analyze the data to better understand and improve its business.

In addition to a relational database, a data warehouse environment can include:

- An extraction, transportation, transformation, and loading (ETL) solution
- Statistical analysis
- Reporting
- Data mining capabilities
- Client analysis tools
- Applications that manage the process of gathering data; transforming it into useful, actionable information; and delivering it to business users

Data warehousing systems typically:

- Use many indexes
- Use some (but not many) joins
- Use denormalized or partially denormalized schemas (such as a star schema) to optimize query and analytical performance
- Use derived data and aggregates
- Have workloads designed to accommodate ad hoc queries and data analysis

  Because you might not know the workload of your data warehouse in advance, you must optimize the data warehouse to perform well for a wide variety of possible query and analytical operations.
- Are updated regularly (nightly or weekly) by the ETL process, using bulk data modification techniques

The end users of a data warehouse do not directly update the data warehouse except when using analytical tools (such as data mining) to make predictions with associated probabilities, assign customers to market segments, and develop customer profiles.

See Also:

Oracle Database Data Warehousing Guide for information about data warehousing, including a comparison with online transaction processing (OLTP)

1.8.2 Online Transaction Processing (OLTP)

Online transaction processing (OLTP) systems are optimized for fast and reliable transaction handling. Compared to data warehouse systems, most OLTP interactions involve a relatively small number of rows, but a larger group of tables. In OLTP systems, performance requirements require that historical data be frequently moved to an archive.

OLTP systems typically:

- Use few indexes.
- Use many joins.
- Use fully normalized schemas to optimize update, insert, and delete performance, and to guarantee data consistency.
- Rarely use derived data and aggregates.
- Have workloads consisting of predefined operations.
- Have users routinely issuing individual data modification statements to the database, so that the OLTP database always reflects the current state of each transaction.

See Also:

Oracle Database Concepts for more information including links to manuals with detailed information

1.9 Features for Special Scenarios

This topic introduces Oracle Database features that are particularly useful in scenarios that involve very large databases and the need for high performance.

Topics:

- SQL Analytic Functions
- Materialized Views
- Partitioning
- Temporal Validity Support

1.9.1 SQL Analytic Functions

A **SQL analytic function** computes an aggregate value based on a group of rows. A SQL analytic function differs from an aggregate function in that it returns multiple rows for each group. For each row, a window of rows is defined. The window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on either a physical number of rows or a logical interval such as time.

SQL analytic functions bring to set-oriented SQL the ability to use array semantics on result sets. They provide coding efficiency, because they enable concise, straightforward coding of logic that is otherwise cumbersome or impossible. They also provide processing efficiency, because they are integral to Oracle Database and use internal optimizations.

A typical use of analytic functions is to retrieve the most current information in a table. For example, a query of the following form returns information from the row with the most recent update time for each customer with records in a table:

```
SELECT ... FROM my_table t1
  WHERE upd_time = ( SELECT MAX(UPD _TIME)
                     FROM my_table t2
                     WHERE t2.cust_id = t1.cust_id );
```

The preceding query uses a correlated subquery to find the `MAX(UPD _TIME)` by `cust _id`, record by record. Therefore, the correlated subquery could be evaluated once for each row in the table. If the table has very few records, performance may be adequate; if the table has tens of thousands of records, the cumulative cost of repeatedly executing the correlated subquery is high.

The following query makes a single pass on the table and computes the maximum `UPD_TIME` during that pass. Depending on various factors, such as table size and number of rows returned, the following query may be much more efficient than the preceding query:

```
SELECT ...
  FROM ( SELECT t1.*,
         MAX(UPD_TIME) OVER (PARTITION BY cust _id) max_time
         FROM my_table t1
       )
  WHERE  upd_time = max_time;
```

The available analytic functions are:

```
AVG
CORR
COUNT
COVAR_POP
COVAR_SAMP
CUME_DIST
DENSE_RANK
FIRST
FIRST_VALUE
LAG
LAST
LAST_VALUE
LEAD
LISTAGG
MAX
MIN
NTH_VALUE
NTILE
PERCENT_RANK
```

```
PERCENTILE_CONT
PERCENTILE_DISC
RANK
RATIO_TO_REPORT
REGR_ (Linear Regression) Functions
ROW_NUMBER
STDDEV
STDDEV_POP
STDDEV_SAMP
SUM
VAR_POP
VAR_SAMP
VARIANCE
```

See Also:

- *Oracle Database Concepts* for an overview of SQL analytic functions
- *Oracle Database SQL Language Reference* for syntax and reference information
- *Oracle Database Data Warehousing Guide* for an extensive discussion of SQL for analysis and reporting

1.9.2 Materialized Views

Materialized views are query results that have been stored ("materialized") as schema objects. Like tables and views, materialized views can appear in the `FROM` clauses of queries.

Materialized views are used to summarize, compute, replicate, and distribute data. They are useful for pre-answering general classes of questions—users can query the materialized views instead of individually aggregating detail records. Some environments where materialized views are useful are data warehousing, replication, and mobile computing.

Materialized views require time to create and update, and disk space for storage, but these costs are offset by dramatically faster queries. In these respects, materialized views are like indexes, and they are called "the indexes of your data warehouse." Unlike indexes, materialized views can be queried directly (with `SELECT` statements) and sometimes updated with DML statements (depending on the type of update needed).

A major benefit of creating and maintaining materialized views is the ability to take advantage of **query rewrite**, which transforms a SQL statement expressed in terms of tables or views into a statement accessing one or more materialized views that are defined on the detail tables. The transformation is transparent to the end user or application, requiring no intervention and no reference to the materialized view in the SQL statement. Because query rewrite is transparent, materialized views can be added or dropped like indexes without invalidating the SQL in the application code.

The following statement creates and populates a materialized aggregate view based on three primary tables in the SH sample schema:

```
CREATE MATERIALIZED VIEW sales_mv AS
  SELECT t.calendar_year, p.prod_id, SUM(s.amount_sold) AS sum_sales
  FROM   times t, products p, sales s
  WHERE  t.time_id = s.time_id
  AND    p.prod_id = s.prod_id
  GROUP BY t.calendar_year, p.prod_id;
```

See Also:

- *Oracle Database Concepts* for an overview of materialized views
- *Oracle Database Data Warehousing Guide* to learn how to use materialized views, including the use of query rewrite with materialized views, in a data warehouse
- Partitioning for more information about how to partition materialized views like tables.

1.9.3 Partitioning

Partitioning is the database ability to physically break a very large table, index, or materialized view into smaller pieces that it can manage independently. Partitioning is similar to parallel processing, which breaks a large process into smaller pieces that can be processed independently.

Each partition is an independent object with its own name and, optionally, its own storage characteristics. Partitioning is useful for many different types of database applications, particularly those that manage large volumes of data. Benefits include increased availability, easier administration of schema objects, reduced contention for shared resources in OLTP systems, and enhanced query performance in data warehouses.

To partition a table, specify the `PARTITION BY` clause in the `CREATE TABLE` statement. `SELECT` and DML statements do not need special syntax to benefit from the partitioning.

A common strategy is to partition records by date ranges. The following statement creates four partitions, one for records from each of four years of sales data (2008 through 2011):

```
CREATE TABLE time_range_sales
   ( prod_id        NUMBER(6)
   , cust_id        NUMBER
   , time_id        DATE
   , channel_id     CHAR(1)
   , promo_id       NUMBER(6)
   , quantity_sold  NUMBER(3)
   , amount_sold    NUMBER(10,2)
   )
PARTITION BY RANGE (time_id)
 (PARTITION SALES_2008 VALUES LESS THAN (TO_DATE('01-JAN-2009','DD-MON-YYYY')),
  PARTITION SALES_2009 VALUES LESS THAN (TO_DATE('01-JAN-2010','DD-MON-YYYY')),
  PARTITION SALES_2010 VALUES LESS THAN (TO_DATE('01-JAN-2011','DD-MON-YYYY')),
  PARTITION SALES_2011 VALUES LESS THAN (MAXVALUE)
 );
```

See Also:

- *Oracle Database Concepts* for an overview of partitions
- *Oracle Database VLDB and Partitioning Guide* for detailed explanations and usage information about partitioning with very large databases (VLDB)

1.9.4 Temporal Validity Support

Temporal Validity Support lets you associate one or more valid time dimensions with a table and have data be visible depending on its time-based validity, as determined by the start and end dates or time stamps of the period for which a given record is considered valid. Examples of time-based validity are the hire and termination dates of an employee in a Human Resources application, the effective date of coverage for an insurance policy, and the effective date of a change of address for a customer or client.

Temporal Validity Support is typically used with Oracle Flashback Technology, for queries that specify the valid time period in `AS OF` and `VERSIONS BETWEEN` clauses. You can also use the `DBMS_FLASHBACK_ARCHIVE.ENABLE_AT_VALID_TIME` procedure to specify an option for the visibility of table data: all table data (the default), data valid at a specified time, or currently valid data within the valid time period at the session level.

Temporal Validity Support is useful in Information Lifecycle Management (ILM) and any other application where it is important to know when certain data becomes valid (from the application's perspective) and when it becomes invalid (if ever).

Note:

Creating and using a table with valid time support and changing data using Temporal Validity Support assume that the user has privileges to create tables and perform data manipulation language (DML) and (data definition language) DDL operations on them.

Example 1-1 Creating and Using a Table with Valid Time Support

The following example creates a table with Temporal Validity Support, inserts rows, and issues queries whose results depend on the valid start date and end date for individual rows.

```
CREATE TABLE my_emp(
  empno NUMBER,
  last_name VARCHAR2(30),
  start_time TIMESTAMP,
  end_time TIMESTAMP,
PERIOD FOR user_valid_time (start_time, end_time));

INSERT INTO my_emp VALUES (100, 'Ames', '01-Jan-10', '30-Jun-11');
INSERT INTO my_emp VALUES (101, 'Burton', '01-Jan-11', '30-Jun-11');
INSERT INTO my_emp VALUES (102, 'Chen', '01-Jan-12', null);

-- Valid Time Queries --

-- AS OF PERIOD FOR queries:

-- Returns only Ames.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Jun-10');

-- Returns  Ames and Burton, but not Chen.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Jun-11');

-- Returns no one.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP( '01-Jul-11');

-- Returns only Chen.
```

```
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Feb-12');

-- VERSIONS PERIOD FOR ... BETWEEN queries:

-- Returns only Ames.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
   TO_TIMESTAMP('01-Jun-10') AND TO_TIMESTAMP('02-Jun-10');

-- Returns Ames and Burton.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
   TO_TIMESTAMP('01-Jun-10') AND TO_TIMESTAMP('01-Mar-11');

-- Returns only Chen.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
   TO_TIMESTAMP('01-Nov-11') AND TO_TIMESTAMP('01-Mar-12');

-- Returns no one.
SELECT * from my_emp VERSIONS PERIOD FOR user_valid_time BETWEEN
   TO_TIMESTAMP('01-Jul-11') AND TO_TIMESTAMP('01-Sep-11');
```

To add Temporal Validity Support to an existing table without explicitly adding columns, use the `ALTER TABLE` statement with the `ADD PERIOD FOR` clause. For example, if the `CREATE TABLE` statement did not create the `START_TIME` and `END_TIME` columns, you could use the following statement to create the same:

```
ALTER TABLE my_emp ADD PERIOD FOR user_valid_time;
```

The preceding statement adds two hidden columns to the table `MY_EMP`: `USER_VALID_TIME_START` and `USER_VALID_TIME_END`. You can insert rows that specify values for these columns, but the columns do not appear in the output of the SQL*Plus `DESCRIBE` statement, and `SELECT` statements show the data in those columns only if the `SELECT` list explicitly includes the column names.

Example 1-2 uses Temporal Validity Support for data change in the table created in Example 1-1. In Example 1-2, the initial record for employee 103 has the last name Davis. Later, the employee changes the last name to Smith. The `END_TIME` value in the original row changes from `NULL` to the day before the change is to become valid. A new row is inserted with the new last name, the appropriate `START_TIME` value, and `END_TIME` set to `NULL` to indicate that it is valid until set otherwise.

Example 1-2   Data Change Using Temporal Validity Support

```
-- Add a record for Davis.
INSERT INTO my_emp VALUES (103, 'Davis', '01-Jan-12', null);

-- To change employee 103's last name to Smith,
-- first set an end date for the record with the old name.
UPDATE my_emp SET end_time = '01-Feb-12' WHERE empno = 103;

-- Then insert another record for employee 103, specifying the new last name,
-- the appropriate valid start date, and null for the valid end date.
-- After the INSERT statement, there are two records for #103 (Davis and Smith).
INSERT INTO my_emp VALUES (103, 'Smith', '02-Feb-12', null);

-- What's the valid information for employee 103 as of today?
SELECT * from my_emp AS OF PERIOD FOR user_valid_time SYSDATE WHERE empno = 103;

-- What was the valid information for employee 103 as of a specified date?

-- First, as of a date after the change to Smith.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('01-Jul-12')
```

```
  WHERE empno = 103;

-- Next, as of a date before the change to Smith.
SELECT * from my_emp AS OF PERIOD FOR user_valid_time TO_TIMESTAMP('20-Jan-12')
  WHERE empno = 103;
```

Related Links:

- `CREATE TABLE`, `ALTER TABLE` , and `SELECT`
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database VLDB and Partitioning Guide*
- Using Oracle Flashback Technology
- Multitenant Container Database Restrictions for Oracle Flashback Technology