33

Developing Applications for a Distributed Database System

Developing applications for a distributed database system includes tasks such as managing the distribution of application data, controlling connections established by database links, maintaining referential integrity, tuning distributed queries, and handling errors in remote procedures.

- Managing the Distribution of Application Data
  In a distributed database environment, coordinate with the database administrator to determine the best location for the data.
- Controlling Connections Established by Database Links
  When a global object name is referenced in a SQL statement or remote procedure call, database links establish a connection to a session in the remote database on behalf of the local user.
- Maintaining Referential Integrity in a Distributed System
  Design your application to check for any returned error messages that indicate that a portion of the distributed update has failed. If you detect a failure, then you should roll back the entire transaction before allowing the application to proceed.
- Tuning Distributed Queries
  The local Oracle Database server breaks the distributed query into a corresponding number of remote queries, which it then sends to the remote nodes for execution. The remote nodes execute the queries and send the results back to the local node. The local node then performs any necessary post-processing and returns the results to the user or application.
- Handling Errors in Remote Procedures
  Errors can occur when a database executes a procedure.

> **See Also:**
>
> *Oracle Database Development Guide* for more information about application development in an Oracle Database environment

33.1 Managing the Distribution of Application Data

In a distributed database environment, coordinate with the database administrator to determine the best location for the data.

Some issues to consider are:

- Number of transactions posted from each location
- Amount of data (portion of table) used by each node
- Performance characteristics and reliability of the network

- Speed of various nodes, capacities of disks
- Importance of a node or link when it is unavailable
- Need for referential integrity among tables

33.2 Controlling Connections Established by Database Links

When a global object name is referenced in a SQL statement or remote procedure call, database links establish a connection to a session in the remote database on behalf of the local user.

The remote connection and session are only created if the connection has not already been established previously for the local user session.

The connections and sessions established to remote databases persist for the duration of the local user's session, unless the application or user explicitly terminates them. Note that when you issue a `SELECT` statement across a database link, a transaction lock is placed on the undo segments. To rerelease the segment, you must issue a `COMMIT` or `ROLLBACK` statement.

Terminating remote connections established using database links is useful for disconnecting high cost connections that are no longer required by the application. You can terminate a remote connection and session using the `ALTER SESSION` statement with the `CLOSE DATABASE LINK` clause. For example, assume you issue the following transactions:

```
SELECT * FROM emp@sales;
COMMIT;
```

The following statement terminates the session in the remote database pointed to by the `sales` database link:

```
ALTER SESSION CLOSE DATABASE LINK sales;
```

To close a database link connection in your user session, you must have the `ALTER SESSION` system privilege.

> **Note:**
>
> Before closing a database link, first close all cursors that use the link and then end your current transaction if it uses the link.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` statement

33.3 Maintaining Referential Integrity in a Distributed System

Design your application to check for any returned error messages that indicate that a portion of the distributed update has failed. If you detect a failure, then you should roll back the entire transaction before allowing the application to proceed.

If a part of a distributed statement fails, for example, due to an integrity constraint violation, the database returns error number `ORA-02055`. Subsequent statements or procedure calls return error number `ORA-02067` until a `ROLLBACK` or `ROLLBACK TO SAVEPOINT` is issued.

The database does not permit declarative referential integrity constraints to be defined across nodes of a distributed system. In other words, a declarative referential integrity constraint on one table cannot specify a foreign key that references a primary or unique key of a remote table. Nevertheless, you can maintain parent/child table relationships across nodes using triggers.

If you decide to define referential integrity across the nodes of a distributed database using triggers, be aware that network failures can limit the accessibility of not only the parent table, but also the child table. For example, assume that the child table is in the `sales` database and the parent table is in the `hq` database. If the network connection between the two databases fails, some DML statements against the child table (those that insert rows into the child table or update a foreign key value in the child table) cannot proceed because the referential integrity triggers must have access to the parent table in the `hq` database.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for more information about using triggers to enforce referential integrity

33.4 Tuning Distributed Queries

The local Oracle Database server breaks the distributed query into a corresponding number of remote queries, which it then sends to the remote nodes for execution. The remote nodes execute the queries and send the results back to the local node. The local node then performs any necessary post-processing and returns the results to the user or application.

> **Note:**
>
> SQL management objects, such as SQL profiles, SQL plan baselines, and SQL patches, and stored outlines might not always work as expected if your query references remote tables with database links. For example, for SQL plan management, when Oracle uses a SQL plan baseline for the query, the parts of the query that are remotely executed might use a different plan than when the SQL plan baseline was created.

- Using Collocated Inline Views
  The most effective way of optimizing distributed queries is to access the remote databases as little as possible and to retrieve only the required data.
- Using Cost-Based Optimization
  Using cost-based optimization includes completing tasks such as rewriting queries and setting up cost-based optimization.
- Using Hints
  Hints can extend the capability of cost-based optimization.
- Analyzing the Execution Plan
  An important aspect to tuning distributed queries is analyzing the execution plan.

33.4.1 Using Collocated Inline Views

The most effective way of optimizing distributed queries is to access the remote databases as little as possible and to retrieve only the required data.

For example, assume you reference five remote tables from two different remote databases in a distributed query and have a complex filter (for example, `WHERE r1.salary + r2.salary > 50000`). You can improve the performance of the query by rewriting the query to access the remote databases once and to apply the filter at the remote site. This rewrite causes less data to be transferred to the query execution site.

Rewriting your query to access the remote database once is achieved by using collocated inline views. The following terms need to be defined:

- **Collocated**

  Two or more tables located in the same database.

- **Inline view**

  A `SELECT` statement that is substituted for a table in a parent `SELECT` statement. The embedded `SELECT` statement, shown within the parentheses is an example of an inline view:

  ```
  SELECT e.empno,e.ename,d.deptno,d.dname
     FROM (SELECT empno, ename from
           emp@orc1.world) e, dept d;
  ```

- **Collocated inline view**

  An inline view that selects data from multiple tables from a single database only. It reduces the amount of times that the remote database is accessed, improving the performance of a distributed query.

Oracle recommends that you form your distributed query using collocated inline views to increase the performance of your distributed query. Oracle Database cost-based optimization can transparently rewrite many of your distributed queries to take advantage of the performance gains offered by collocated inline views.

33.4.2 Using Cost-Based Optimization

Using cost-based optimization includes completing tasks such as rewriting queries and setting up cost-based optimization.

- How Does Cost-Based Optimization Work?
  The main task of optimization is to rewrite a distributed query to use collocated inline views.
- Rewriting Queries for Cost-Based Optimization
  In addition to rewriting your queries with collocated inline views, the cost-based optimization method optimizes distributed queries according to the gathered statistics of the referenced tables and the computations performed by the optimizer.
- Setting Up Cost-Based Optimization
  After you have set up your system to use cost-based optimization to improve the performance of distributed queries, the operation is transparent to the user. In other words, the optimization occurs automatically when the query is issued.

33.4.2.1 How Does Cost-Based Optimization Work?

The main task of optimization is to rewrite a distributed query to use collocated inline views.

This optimization is performed in three steps:

1. All mergeable views are merged.
2. Optimizer performs collocated query block test.
3. Optimizer rewrites query using collocated inline views.

After the query is rewritten, it is executed and the data set is returned to the user.

While cost-based optimization is performed transparently to the user, it cannot improve the performance of several distributed query scenarios. Specifically, if your distributed query contains any of the following, cost-based optimization is not effective:

- Aggregates
- Subqueries
- Complex SQL

If your distributed query contains one of these elements, see "Using Hints" to learn how you can modify your query and use hints to improve the performance of your distributed query.

33.4.2.2 Rewriting Queries for Cost-Based Optimization

In addition to rewriting your queries with collocated inline views, the cost-based optimization method optimizes distributed queries according to the gathered statistics of the referenced tables and the computations performed by the optimizer.

For example, cost-based optimization analyzes the following query. The example assumes that table statistics are available. Note that it analyzes the query inside a `CREATE TABLE` statement:

```
CREATE TABLE AS (
                 SELECT l.a, l.b, r1.c, r1.d, r1.e, r2.b, r2.c
                 FROM local l, remote1 r1, remote2 r2
                    WHERE l.c = r.c
                    AND r1.c = r2.c
                    AND r.e > 300
                );
```

and rewrites it as:

```
CREATE TABLE AS (
                 SELECT l.a, l.b, v.c, v.d, v.e
                 FROM (
                        SELECT r1.c, r1.d, r1.e, r2.b, r2.c
                         FROM remote1 r1, remote2 r2
                         WHERE r1.c = r2.c
                         AND r1.e > 300
                      ) v, local l
                 WHERE l.c = r1.c
                );
```

The alias `v` is assigned to the inline view, which can then be referenced as a table in the preceding `SELECT` statement. Creating a collocated inline view reduces the amount of queries performed at a remote site, thereby reducing costly network traffic.

33.4.2.3 Setting Up Cost-Based Optimization

After you have set up your system to use cost-based optimization to improve the performance of distributed queries, the operation is transparent to the user. In other words, the optimization occurs automatically when the query is issued.

- Setting Up the Environment
  Set the `OPTIMIZER_MODE` initialization parameter to establish the default behavior for choosing an optimization approach for the instance.
- Analyzing Tables
  For cost-based optimization to select the most efficient path for a distributed query, you must provide accurate statistics for the tables involved. You do this using the `DBMS_STATS` package.

33.4.2.3.1 Setting Up the Environment

Set the `OPTIMIZER_MODE` initialization parameter to establish the default behavior for choosing an optimization approach for the instance.

You can set this parameter by:

- Modifying the `OPTIMIZER_MODE` parameter in the initialization parameter file
- Setting it at session level by issuing an `ALTER SESSION` statement

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information on setting the `OPTIMIZER_MODE` initialization parameter in the parameter file and for configuring your system to use a cost-based optimization method

33.4.2.3.2 Analyzing Tables

For cost-based optimization to select the most efficient path for a distributed query, you must provide accurate statistics for the tables involved. You do this using the `DBMS_STATS` package.

> **Note:**
>
> You must connect locally with respect to the tables when executing the `DBMS_STATS` procedure.
>
> You must first connect to the remote site and then execute a `DBMS_STATS` procedure.

The following `DBMS_STATS` procedures enable the gathering of certain classes of optimizer statistics:

- `GATHER_INDEX_STATS`
- `GATHER_TABLE_STATS`
- `GATHER_SCHEMA_STATS`

- `GATHER_DATABASE_STATS`

For example, assume that distributed transactions routinely access the `scott.dept` table. To ensure that the cost-based optimizer is still picking the best plan, execute the following:

```
BEGIN
   DBMS_STATS.GATHER_TABLE_STATS ('scott', 'dept');
END;
```

See Also:

- *Oracle Database SQL Tuning Guide* for information about generating statistics
- *Oracle Database PL/SQL Packages and Types Reference* for additional information on using the `DBMS_STATS` package

33.4.3 Using Hints

Hints can extend the capability of cost-based optimization.

- About Using Hints
  If a statement is not sufficiently optimized, then you can use hints to extend the capability of cost-based optimization. Specifically, if you write your own query to use collocated inline views, instruct the cost-based optimizer not to rewrite your distributed query.
- Using the NO_MERGE Hint
  The `NO_MERGE` hint prevents the database from merging an inline view into a potentially non-collocated SQL statement.
- Using the DRIVING_SITE Hint
  The `DRIVING_SITE` hint lets you specify the site where the query execution is performed.

33.4.3.1 About Using Hints

If a statement is not sufficiently optimized, then you can use hints to extend the capability of cost-based optimization. Specifically, if you write your own query to use collocated inline views, instruct the cost-based optimizer not to rewrite your distributed query.

Additionally, if you have special knowledge about the database environment (such as statistics, load, network and CPU limitations, distributed queries, and so forth), you can specify a hint to guide cost-based optimization. For example, if you have written your own optimized query using collocated inline views that are based on your knowledge of the database environment, specify the `NO_MERGE` hint to prevent the optimizer from rewriting your query.

This technique is especially helpful if your distributed query contains an aggregate, subquery, or complex SQL. Because this type of distributed query cannot be rewritten by the optimizer, specifying `NO_MERGE` causes the optimizer to skip the steps described in "How Does Cost-Based Optimization Work?".

The `DRIVING_SITE` hint lets you define a remote site to act as the query execution site. In this way, the query executes on the remote site, which then returns the data to the local site. This hint is especially helpful when the remote site contains the majority of the data.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about using hints

33.4.3.2 Using the NO_MERGE Hint

The `NO_MERGE` hint prevents the database from merging an inline view into a potentially non-collocated SQL statement.

This hint is embedded in the `SELECT` statement and can appear either at the beginning of the `SELECT` statement with the inline view as an argument or in the query block that defines the inline view.

```
/* with argument */

SELECT /*+NO_MERGE(v)*/ t1.x, v.avg_y
   FROM t1, (SELECT x, AVG(y) AS avg_y FROM t2 GROUP BY x) v,
   WHERE t1.x = v.x AND t1.y = 1;

/* in query block */

SELECT t1.x, v.avg_y
   FROM t1, (SELECT /*+NO_MERGE*/ x, AVG(y) AS avg_y FROM t2 GROUP BY x) v,
   WHERE t1.x = v.x AND t1.y = 1;
```

Typically, you use this hint when you have developed an optimized query based on your knowledge of your database environment.

Related Topics

- Using Hints
  Hints can extend the capability of cost-based optimization.

33.4.3.3 Using the DRIVING_SITE Hint

The `DRIVING_SITE` hint lets you specify the site where the query execution is performed.

It is best to let cost-based optimization determine where the execution should be performed, but if you prefer to override the optimizer, you can specify the execution site manually.

Following is an example of a `SELECT` statement with a `DRIVING_SITE` hint:

```
SELECT /*+DRIVING_SITE(dept)*/ * FROM emp, dept@remote.com
   WHERE emp.deptno = dept.deptno;
```

Related Topics

- Using Hints
  Hints can extend the capability of cost-based optimization.

33.4.4 Analyzing the Execution Plan

An important aspect to tuning distributed queries is analyzing the execution plan.

The feedback that you receive from your analysis is an important element to testing and verifying your database. Verification becomes especially important when you want to compare plans. For example, comparing the execution plan for a distributed query optimized by cost-

based optimization to a plan for a query manually optimized using hints, collocated inline views, and other techniques.

- Generating the Execution Plan
  After you have prepared the database to store the execution plan, you are ready to view the plan for a specified query. Instead of directly executing a SQL statement, append the statement to the `EXPLAIN PLAN FOR` clause.
- Viewing the Execution Plan
  After you have executed the preceding SQL statement, the execution plan is stored temporarily in the `PLAN_TABLE`.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for detailed information about execution plans, the `EXPLAIN PLAN` statement, and how to interpret the results

33.4.4.1 Generating the Execution Plan

After you have prepared the database to store the execution plan, you are ready to view the plan for a specified query. Instead of directly executing a SQL statement, append the statement to the `EXPLAIN PLAN FOR` clause.

For example, you can execute the following:

```
EXPLAIN PLAN FOR
   SELECT d.dname
   FROM dept d
      WHERE d.deptno
      IN (SELECT deptno
          FROM emp@orc2.world
          GROUP BY deptno
          HAVING COUNT (deptno) >3
          )
/
```

33.4.4.2 Viewing the Execution Plan

After you have executed the preceding SQL statement, the execution plan is stored temporarily in the `PLAN_TABLE`.

To view the results of the execution plan, execute the following script:

```
@utlxpls.sql
```

> **Note:**
>
> The `utlxpls.sql` can be found in the `$ORACLE_HOME/rdbms/admin` directory.

Executing the `utlxpls.sql` script displays the execution plan for the `SELECT` statement that you specified. The results are formatted as follows:

```
Plan Table
--------------------------------------------------------------------------------
| Operation                 |  Name    |  Rows | Bytes|  Cost  | Pstart| Pstop |
--------------------------------------------------------------------------------
| SELECT STATEMENT          |          |       |      |        |       |       |
|  NESTED LOOPS             |          |       |      |        |       |       |
|   VIEW                    |          |       |      |        |       |       |
|    REMOTE                 |          |       |      |        |       |       |
|   TABLE ACCESS BY INDEX RO|DEPT      |       |      |        |       |       |
|    INDEX UNIQUE SCAN      |PK_DEPT   |       |      |        |       |       |
--------------------------------------------------------------------------------
```

If you are manually optimizing distributed queries by writing your own collocated inline views or using hints, it is best to generate an execution plan before and after your manual optimization. With both execution plans, you can compare the effectiveness of your manual optimization and make changes as necessary to improve the performance of the distributed query.

To view the SQL statement that will be executed at the remote site, execute the following select statement:

```
SELECT OTHER
FROM PLAN_TABLE
   WHERE operation = 'REMOTE';
```

Following is sample output:

```
SELECT DISTINCT "A1"."DEPTNO" FROM "EMP" "A1"
   GROUP BY "A1"."DEPTNO" HAVING COUNT("A1"."DEPTNO")>3
```

> **Note:**
>
> If you are having difficulty viewing the entire contents of the `OTHER` column, execute the following SQL*Plus command:
>
> ```
> SET LONG 9999999
> ```

33.5 Handling Errors in Remote Procedures

Errors can occur when a database executes a procedure.

When the database executes a procedure locally or at a remote location, four types of exceptions can occur:

- PL/SQL user-defined exceptions, which must be declared using the keyword `EXCEPTION`
- PL/SQL predefined exceptions such as the `NO_DATA_FOUND` keyword
- SQL errors such as `ORA-00900` and `ORA-02015`
- Application exceptions generated using the `RAISE_APPLICATION_ERROR`() procedure

When using local procedures, you can trap these messages by writing an exception handler such as the following

```
BEGIN
  ...
EXCEPTION
   WHEN ZERO_DIVIDE THEN
```

```
   /* ... handle the exception */
END;
```

Notice that the `WHEN` clause requires an exception name. If the exception does not have a name, for example, exceptions generated with `RAISE_APPLICATION_ERROR`, you can assign one using `PRAGMA_EXCEPTION_INIT`. For example:

```
DECLARE
  null_salary EXCEPTION;
  PRAGMA EXCEPTION_INIT(null_salary, -20101);
BEGIN
  ...
  RAISE_APPLICATION_ERROR(-20101, 'salary is missing');
...
EXCEPTION
  WHEN null_salary THEN
  ...
END;
```

When calling a remote procedure, exceptions can be handled by an exception handler in the local procedure. The remote procedure must return an error number to the local, calling procedure, which then handles the exception as shown in the previous example. Note that PL/SQL user-defined exceptions always return `ORA-06510` to the local procedure.

Therefore, it is not possible to distinguish between two different user-defined exceptions based on the error number. All other remote exceptions can be handled in the same manner as local exceptions.

See Also:

Oracle Database PL/SQL Language Reference for more information about PL/SQL procedures