1

Introduction to SQL Tuning

SQL tuning is the attempt to diagnose and repair SQL statements that fail to meet a performance standard.

About SQL Tuning

SQL tuning is the iterative process of improving SQL statement performance to meet specific, measurable, and achievable goals.

SQL tuning implies fixing problems in deployed applications. In contrast, application design sets the security and performance goals *before* deploying an application.

See Also:

- SQL Performance Methodology
- "Guidelines for Designing Your Application" to learn how to design for SQL performance

Purpose of SQL Tuning

A SQL statement becomes a problem when it fails to perform according to a predetermined and measurable standard.

After you have identified the problem, a typical tuning session has one of the following goals:

- Reduce user response time, which means decreasing the time between when a user issues a statement and receives a response
- Improve throughput, which means using the least amount of resources necessary to process all rows accessed by a statement

For a response time problem, consider an online book seller application that hangs for three minutes after a customer updates the shopping cart. Contrast with a three-minute parallel query in a data warehouse that consumes all of the database host CPU, preventing other queries from running. In each case, the user response time is three minutes, but the cause of the problem is different, and so is the tuning goal.

Prerequisites for SQL Tuning

SQL performance tuning requires a foundation of database knowledge.

If you are tuning SQL performance, then this manual assumes that you have the knowledge and skills shown in the following table.

Table 1-1 Required Knowledge

Required Knowledge	Description	To Learn More
Database architecture	Database architecture is not the domain of administrators alone. As a developer, you want to develop applications in the least amount of time against an Oracle database, which requires exploiting the database architecture and features. For example, not understanding Oracle Database concurrency controls and multiversioning read consistency may make an application corrupt the integrity of the data, run slowly, and decrease scalability.	*Oracle Database Concepts* explains the basic relational data structures, transaction management, storage structures, and instance architecture of Oracle Database.
SQL and PL/SQL	Because of the existence of GUI-based tools, it is possible to create applications and administer a database without knowing SQL. However, it is impossible to tune applications or a database without knowing SQL.	*Oracle Database Concepts* includes an introduction to Oracle SQL and PL/SQL. You must also have a working knowledge of *Oracle Database SQL Language Reference*, *Oracle Database PL/SQL Packages and Types Reference*, and *Oracle Database PL/SQL Packages and Types Reference*.
SQL tuning tools	The database generates performance statistics, and provides SQL tuning tools that interpret these statistics.	*Oracle Database Get Started with Performance Tuning* provides an introduction to the principal SQL tuning tools.

Tasks and Tools for SQL Tuning

After you have identified the goal for a tuning session, for example, reducing user response time from three minutes to less than a second, the problem becomes how to accomplish this goal.

SQL Tuning Tasks

The specifics of a tuning session depend on many factors, including whether you tune proactively or reactively.

In **proactive SQL tuning**, you regularly use SQL Tuning Advisor to determine whether you can make SQL statements perform better. In **reactive SQL tuning**, you correct a SQL-related problem that a user has experienced.

Whether you tune proactively or reactively, a typical SQL tuning session involves all or most of the following tasks:

1. Identifying high-load SQL statements

   Review past execution history to find the statements responsible for a large share of the application workload and system resources.

2. Gathering performance-related data

   The optimizer statistics are crucial to SQL tuning. If these statistics do not exist or are no longer accurate, then the optimizer cannot generate the best plan. Other data relevant to

SQL performance include the structure of tables and views that the statement accessed, and definitions of any indexes available to the statement.

3. Determining the causes of the problem

   Typically, causes of SQL performance problems include:

   - Inefficiently designed SQL statements

     If a SQL statement is written so that it performs unnecessary work, then the optimizer cannot do much to improve its performance. Examples of inefficient design include

     – Neglecting to add a join condition, which leads to a Cartesian join

     – Using hints to specify a large table as the driving table in a join

     – Specifying `UNION` instead of `UNION ALL`

     – Making a subquery execute for every row in an outer query

   - Suboptimal execution plans

     The query optimizer (also called the optimizer) is internal software that determines which execution plan is most efficient. Sometimes the optimizer chooses a plan with a suboptimal access path, which is the means by which the database retrieves data from the database. For example, the plan for a query predicate with low selectivity may use a full table scan on a large table instead of an index.

     You can compare the execution plan of an optimally performing SQL statement to the plan of the statement when it performs suboptimally. This comparison, along with information such as changes in data volumes, can help identify causes of performance degradation.

   - Missing SQL access structures

     Absence of SQL access structures, such as indexes and materialized views, is a typical reason for suboptimal SQL performance. The optimal set of access structures can improve SQL performance by orders of magnitude.

   - Stale optimizer statistics

     Statistics gathered by `DBMS_STATS` can become stale when the statistics maintenance operations, either automatic or manual, cannot keep up with the changes to the table data caused by DML. Because stale statistics on a table do not accurately reflect the table data, the optimizer can make decisions based on faulty information and generate suboptimal execution plans.

   - Hardware problems

     Suboptimal performance might be connected with memory, I/O, and CPU problems.

4. Defining the scope of the problem

   The scope of the solution must match the scope of the problem. Consider a problem at the database level and a problem at the statement level. For example, the shared pool is too small, which causes cursors to age out quickly, which in turn causes many hard parses. Using an initialization parameter to increase the shared pool size fixes the problem at the database level and improves performance for all sessions. However, if a single SQL statement is not using a helpful index, then changing the optimizer initialization parameters for the entire database could harm overall performance. If a single SQL statement has a problem, then an appropriately scoped solution addresses just this problem with this statement.

5. Implementing corrective actions for suboptimally performing SQL statements

These actions vary depending on circumstances. For example, you might rewrite a SQL statement to be more efficient, avoiding unnecessary hard parsing by rewriting the statement to use bind variables. You might also use equijoins, remove functions from `WHERE` clauses, and break a complex SQL statement into multiple simple statements.

In some cases, you improve SQL performance not by rewriting the statement, but by restructuring schema objects. For example, you might index a new access path, or reorder columns in a concatenated index. You might also partition a table, introduce derived values, or even change the database design.

6. Preventing SQL performance regressions

   To ensure optimal SQL performance, verify that execution plans continue to provide optimal performance, and choose better plans if they come available. You can achieve these goals using optimizer statistics, SQL profiles, and SQL plan baselines.

See Also:

- "Shared Pool Check"
- *Oracle Database Concepts* to learn more about the shared pool

SQL Tuning Tools

SQL tuning tools are either automated or manual.

In this context, a tool is automated if the database itself can provide diagnosis, advice, or corrective actions. A manual tool requires you to perform all of these operations.

All tuning tools depend on the basic tools of the dynamic performance views, statistics, and metrics that the database instance collects. The database itself contains the data and metadata required to tune SQL statements.

Automated SQL Tuning Tools

Oracle Database provides several advisors relevant for SQL tuning.

Additionally, SQL plan management is a mechanism that can prevent performance regressions and also help you to improve SQL performance.

All of the automated SQL tuning tools can use SQL tuning sets as input. A SQL tuning set (STS) is a database object that includes one or more SQL statements along with their execution statistics and execution context.

See Also:

- "About SQL Tuning Sets"
- *Oracle Database Get Started with Performance Tuning* to learn more about managing SQL tuning sets

Automatic Database Diagnostic Monitor (ADDM)

ADDM is self-diagnostic software built into Oracle Database.

ADDM can automatically locate the root causes of performance problems, provide recommendations for correction, and quantify the expected benefits. ADDM also identifies areas where no action is necessary.

ADDM and other advisors use Automatic Workload Repository (AWR), which is an infrastructure that provides services to database components to collect, maintain, and use statistics. ADDM examines and analyzes statistics in AWR to determine possible performance problems, including high-load SQL.

For example, you can configure ADDM to run nightly. In the morning, you can examine the latest ADDM report to see what might have caused a problem and if there is a recommended fix. The report might show that a particular `SELECT` statement consumed a huge amount of CPU, and recommend that you run SQL Tuning Advisor.

See Also:

- *Oracle Database Get Started with Performance Tuning*
- *Oracle Database Performance Tuning Guide*

SQL Tuning Advisor

SQL Tuning Advisor is internal diagnostic software that identifies problematic SQL statements and recommends how to improve statement performance.

When run during database maintenance windows as an automated maintenance task, SQL Tuning Advisor is known as Automatic SQL Tuning Advisor.

SQL Tuning Advisor takes one or more SQL statements as an input and invokes the Automatic Tuning Optimizer to perform SQL tuning on the statements. The advisor performs the following types of analysis:

- Checks for missing or stale statistics
- Builds SQL profiles

  A SQL profile is a set of auxiliary information specific to a SQL statement. A SQL profile contains corrections for suboptimal optimizer estimates discovered during Automatic SQL Tuning. This information can improve optimizer estimates for cardinality, which is the number of rows that is estimated to be or actually is returned by an operation in an execution plan, and selectivity. These improved estimates lead the optimizer to select better plans.
- Explores whether a different access path can significantly improve performance
- Identifies SQL statements that lend themselves to suboptimal plans

The output is in the form of advice or recommendations, along with a rationale for each recommendation and its expected benefit. The recommendation relates to a collection of statistics on objects, creation of new indexes, restructuring of the SQL statement, or creation of a SQL profile. You can choose to accept the recommendations to complete the tuning of the SQL statements.

See Also:

- "Analyzing SQL with SQL Tuning Advisor"
- *Oracle Database Get Started with Performance Tuning*

SQL Access Advisor

SQL Access Advisor is internal diagnostic software that recommends which materialized views, indexes, and materialized view logs to create, drop, or retain.

SQL Access Advisor takes an actual workload as input, or the advisor can derive a hypothetical workload from the schema. SQL Access Advisor considers the trade-offs between space usage and query performance, and recommends the most cost-effective configuration of new and existing materialized views and indexes. The advisor also makes recommendations about partitioning.

See Also:

- "About SQL Access Advisor"
- *Oracle Database Get Started with Performance Tuning*
- *Oracle Database Administrator’s Guide* to learn more about automated indexing
- *Oracle Database Licensing Information User Manual* for details on whether automated indexing is supported for different editions and services

Automatic Indexing

Oracle Database can constantly monitor the application workload, creating and managing indexes automatically.

Note:

See *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services.

Creating indexes manually requires deep knowledge of the data model, application, and data distribution. Often DBAs make choices about which indexes to create, and then never revise their choices. As a result, opportunities for improvement are lost, and unnecessary indexes can become a performance liability. Automatic index management solves this problem.

How Automatic Indexing Works

The automatic indexing process runs in the background every 15 minutes and performs the following operations:

1. Automatic index candidates are identified based on the usage of table columns in SQL statements.

- Tables that have high DML activity are excluded if the performance overhead of index maintenance during DML outweighs the benefit of improved query performance.
- Automatic indexes can be single-column or multi-column.
- Tables with stale or missing optimizer statistics are not considered for automatic indexes until statistics are gathered and fresh.

2. Index candidates are initially created invisible and unusable. At this stage, they are metadata only and not visible to the application workload. Index candidates can be:
   - Table columns (including virtual columns).
   - Partitioned and non-partitioned tables.
   - Selected expressions (for example, JSON expressions).
   - Single or multi-column.
3. A sample of workload SQL statements is test parsed to determine which indexes are determined by the optimizer to be beneficial. Indexes deemed useful by the optimizer are built and made valid so that the performance effect on SQL statements can be measured. All candidate indexes remain invisible during the verification step.

   Indexes that are not identified as useful by the optimizer remain invisible and unusable.
4. A sample of workload SQL statements is test executed to determine which indexes improve performance.
5. Candidate valid indexes found to improve SQL performance will be made visible and available to the application workload. Candidate indexes that do not improve SQL performance will revert to invisible and be unusable after a short delay.

   During the verification stage, if an index is found to be beneficial, but an individual SQL statement suffers a performance regression, a SQL plan baseline is created to prevent the regression when the index is made visible.
6. Unused automatic indexes are dropped by the automatic indexing process after the configurable retention period has expired.

Note:

By default, the unused automatic indexes are deleted after 373 days. The period for retaining the unused automatic indexes in a database can be configured using the `DBMS_AUTO_INDEX.CONFIGURE` procedure.

See Also:

Configuring Automatic Indexing in Oracle Database

Managing Automatic Indexing

The `DBMS_AUTO_INDEX` package provides options for configuring, dropping, monitoring, and reporting on automatic indexing.

Enabling and Configuring Automatic Indexing

Use the `DBMS_AUTO_INDEX.CONFIGURE` procedure to do the following:

- Enable automatic indexing.
  `EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_MODE','IMPLEMENT')`
- Configure additional settings, such as how long to retain unused auto indexes, in days.
  `EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_RETENTION_FOR_AUTO','180')`
- Report on auto index configuration settings.

  ```
  SELECT parameter_name, parameter_value FROM dba_auto_index_config;
  ```

Additional Controls

By setting the OPTIMIZER_SESSION_TYPE initialization parameter to ADHOC in a session, you can suspend automatic indexing for SQL statements in this session. The automatic indexing process does not identify index candidates, or create and verify indexes.

```
ALTER SESSION SET optimizer_session_type = 'ADHOC';
```

See Also:

- *Oracle Database Administrator's Guide* to learn more about automatic indexing
- See DBMS_AUTO_INDEX in the *Oracle Database PL/SQL Packages and Types Reference* to learn about the procedures and functions available in the `DBMS_AUTO_INDEX` package
- *Oracle Database Reference* to learn more about `OPTIMIZER_SESSION_TYPE`.

Dropping Automatic Indexes

`DBMS_AUTO_INDEX.DROP_AUTO_INDEXES` provides three options for dropping automatic indexes. Carefully note the use of single and double quotation marks in the first example.

- Drop a single index owned by a schema and allow recreate.

  ```
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('SH','"SYS_AI_c0cmdvbzgyq94"',TRUE)
  ```

- Drop all indexes owned by a schema and allow recreate.

  ```
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('SH',NULL,TRUE)
  ```

- Drop all indexes owned by a schema and disallow recreate. Then, change the recreation status back to `allow`.

  ```
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('HR',NULL)
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('HR',NULL,TRUE)
  ```

Accounting for DML Overhead

Indexes must be maintained when data is changed. This increases the overhead of DML operations such as `INSERT`, `UPDATE`, and `DELETE`. For example, improvements in query performance may be offset by the cost of DML on tables with significant write activity. Automatic indexes factors in this overhead when deciding whether a new index is beneficial or not. This functionality is controlled by the `CONFIGURE` parameter `AUTO_INDEX_INCLUDE_DML_COST`, which by default is `ON`. The feature can be disabled as follows:

```
EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_INCLUDE_DML_COST','ON')
```

You can use SQL to view the current setting of this parameter:

```
SELECT parameter_name,parameter_value FROM DBA_AUTO_INDEX_CONFIG WHERE
parameter_name = 'AUTO_INDEX_INCLUDE_DML_COST';
```

SQL Plan Management

SQL plan management is a preventative mechanism that enables the optimizer to automatically manage execution plans, ensuring that the database uses only known or verified plans.

This mechanism can build a SQL plan baseline, which contains one or more accepted plans for each SQL statement. By using baselines, SQL plan management can prevent plan regressions from environmental changes, while permitting the optimizer to discover and use better plans.

See Also:

- "Overview of SQL Plan Management"
- *Oracle Database PL/SQL Packages and Types Reference*

  to learn about the `DBMS_SPM` package

How Automatic SQL Plan Management Works

With Automatic SPM (SQL plan management), plan performance regressions are detected and repaired automatically.

When a SQL statement is executed, and the execution plan is new, then after execution the performance of the SQL statement is compared with the lowest cost plan found in ASTS.

When `DBMS_SPM.CONFIGURE` is set to ON:

This configures automatic SPM to use its background verification mode:

```
EXEC DBMS_SPM.CONFIGURE('AUTO_SPM_EVOLVE_TASK', 'ON');
```

In this setting, the high frequency Automatic SPM Evolve Advisor does the following.

- Inspects the Automatic Workload Repository (AWR) and the Automatic SQL Tuning Set (ASTS) to identify SQL statements that consume significant system resources.
- Locates alternative SQL execution plans located in the ASTS.
- Test executes the alternative plans and compares their performance.
- Evaluates which plans perform best and creates SQL plan baselines to enforce them.

When `DBMS_SPM.CONFIGURE` is set to AUTO:

This configures automatic SPM to use real-time mode:

```
EXEC DBMS_SPM.CONFIGURE('AUTO_SPM_EVOLVE_TASK', 'AUTO');
```

The high frequency Automatic SPM Evolve Advisor continues to operate in the background. However, SQL execution plan performance is evaluated immediately in the foreground:

- When a SQL statement is executed, and the execution plan is new, then after execution the performance of the SQL statement is compared with the performance of the plans known and stored in ASTS to check if a more optimal alternative SQL execution plan already exists.
- If a previous plan performs better than the new plan, a SQL plan baseline is created to enforce the more optimal previous plan.

SQL Performance Analyzer

SQL Performance Analyzer determines the effect of a change on a SQL workload by identifying performance divergence for each SQL statement.

System changes such as upgrading a database or adding an index may cause changes to execution plans, affecting SQL performance. By using SQL Performance Analyzer, you can accurately forecast the effect of system changes on SQL performance. Using this information, you can tune the database when SQL performance regresses, or validate and measure the gain when SQL performance improves.

See Also:

Oracle Database Testing Guide

SQL Transpiler

The SQL Transpiler automatically and wherever possible converts (*transpiles*) PL/SQL functions within SQL into SQL expressions, without user intervention.

Expressions in Oracle SQL statements are allowed to call PL/SQL functions. But these calls incur overhead because the PL/SQL runtime must be invoked. The SQL compiler automatically attempts to convert any PL/SQL function called from a SQL statement into a semantically equivalent SQL expression. Transpiling PL/SQL functions into SQL increases performance for new and existing programs and functions. When a transpiled PL/SQL function is invoked, the per row cost of executing the transpiled code within SQL is much lower than switching from the SQL runtime to the PL/SQL runtime in order to execute the original PL/SQL code.

If the transpiler cannot convert a PL/SQL function to SQL, execution of the function falls back to the PL/SQL runtime. Not all PL/SQL constructs are supported by the transpiler.

The entire operation is transparent to users

The following example shows a `SELECT` statement with a call to the PL/SQL function `GET_MONTH_ABBREVIATION`. The function extracts the three letter month abbreviation from the parameter `date_value`. In the Predicate Information section at the bottom of the plan, you can see that `GET_MONTH_ABBREVIATION` is replaced with

`TO_CHAR(INTERNAL_FUNCTION("HIRE_DATE"),'MON','NLS_DATE_LANGUAGE=English')='MAY')`. This indicates that transpilation has occurred.

```
create function get_month_abbreviation (
   date_value date
) return varchar2 is
begin
  return to_char ( date_value, 'MON', 'NLS_DATE_LANGUAGE=English' );
end;
/

alter session set sql_transpiler = ON;

select employee_id, first_name, last_name
from hr.employees
where get_month_abbreviation ( hire_date ) = 'MAY';

EMPLOYEE_ID   FIRST_NAME     LAST_NAME
      ------------------------------------
        104   Bruce          Ernst
        115   Alexander      Khoo
        122   Payam          Kaufling
        174   Ellen          Abel
        178   Kimberely      Grant
        197   Kevin          Feeney

select *
from  dbms_xplan.display_cursor ( format => 'BASIC +PREDICATE' );

---------------------------------------
| Id  | Operation         | Name      |
---------------------------------------
|   0 | SELECT STATEMENT  |           |
|  *1 | TABLE ACCESS FULL | EMPLOYEES |
---------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
    1 -
filter(TO_CHAR(INTERNAL_FUNCTION("HIRE_DATE"),'MON','NLS_DATE_LANGUAGE=English
')='MAY')
```

If transpilation occurs, the database replaces the name of the function in the predicate section with the SQL expression. This is what is happening in the above example.

Note:

If transpilation did not occur, the predicate section in this example would show the following:

```
Predicate Information (identified by operation id):
---------------------------------------------------
    1 - filter("GET_MONTH_ABBREVIATION"("HIRE_DATE")='MAY')
```

Enabling or Disabling the SQL Transpiler

The SQL Transpiler is disabled by default. You can enable it with an ALTER SYSTEM command.

You can change the parameter value using `SQL_TRANSPILER`.

```
SQL_TRANSPILER = [ON | BASIC | OFF]
```

When set to `BASIC`, the transpiler attempts to transpile all functions that only contain a return statement. The parameter can be modified either at the system or session level.

Eligibility of PL/SQL Constructs for Transpilation

Not all PL/SQL constructs can be transpiled to SQL.

PL/SQL Constructs Eligible for Transpilation

The following PL/SQL language elements are supported by the SQL Transpiler:

- Basic SQL scalar types: CHARACTER, DATETIME, and NUMBER
- String types (CHAR, VARCHAR, VARCHAR2, NCHAR, and others.)
- Numeric types (NUMBER, BINARY DOUBLE, and others.)
- Date types (DATE, INTERVAL, and TIMESTAMP)
- Local variables (with optional initialization at declaration) and constants
- Parameters with optional (simple) default values
- Variable assignment statements
- Expressions which can be translated into equivalent SQL expressions
- RETURN statements
- Expressions and local variables of BOOLEAN type
- Functions defined inside of a PL/SQL package.

PL/SQL Constructs not Eligible for Transpilation at This Time

- Embedded SQL statements. A transpiled function cannot contain a cursor declaration, explicit cursors, ref cursors, or an execute-immediate statement
- Package variables, both public and private.
- PL/SQL Specific Scalar Types: PLS_INTEGER

- PL/SQL Aggregate Types: Records, Collections, and Tables
- Oracle Objects (ADT/UDT), XML, and JSON
- Deprecated Datatypes: LONG
- The %TYPE and %ROWTYPE attributes
- Package state (both constants and variables)
- Locally defined PL/SQL types
- Locally defined (nested) functions
- Calls to other PL/SQL functions (both schema-level and package level). This also precludes support for recursive function calls
- Control-flow statements like LOOP, GOTO, and RAISE
- Nested DECLARE-BEGIN-EXCEPTION blocks
- CASE control-flow statements (note that this is different from the SQL CASE expressions which are supported by both SQL and PL/SQL)
- Transaction processing like COMMIT, ROLLBACK, LOCK-TABLE, PRAGMA AUTONOMOUS TRANSACTION, SELECT-FOR-UPDATE, and others

Manual SQL Tuning Tools

In some situations, you may want to run manual tools in addition to the automated tools. Alternatively, you may not have access to the automated tools.

Execution Plans

Execution plans are the principal diagnostic tool in manual SQL tuning. For example, you can view plans to determine whether the optimizer selects the plan you expect, or identify the effect of creating an index on a table.

You can display execution plans in multiple ways. The following tools are the most commonly used:

- `DBMS_XPLAN`

  You can use the `DBMS_XPLAN` package methods to display the execution plan generated by the `EXPLAIN PLAN` command and query of `V$SQL_PLAN`.

- `EXPLAIN PLAN`

  This SQL statement enables you to view the execution plan that the optimizer would use to execute a SQL statement without actually executing the statement. See *Oracle Database SQL Language Reference*.

- `V$SQL_PLAN` and related views

  These views contain information about executed SQL statements, and their execution plans, that are still in the shared pool. See *Oracle Database Reference*.

- `AUTOTRACE`

  The `AUTOTRACE` command in SQL*Plus generates the execution plan and statistics about the performance of a query. This command provides statistics such as disk reads and memory reads. See *SQL*Plus User's Guide and Reference*.

Real-Time SQL Monitoring and Real-Time Database Operations

The Real-Time SQL Monitoring feature of Oracle Database enables you to monitor the performance of SQL statements while they are executing. By default, SQL monitoring starts automatically when a statement runs in parallel, or when it has consumed at least 5 seconds of CPU or I/O time in a single execution.

A database operation is a set of database tasks defined by end users or application code, for example, a batch job or Extraction, Transformation, and Loading (ETL) processing. You can define, monitor, and report on database operations. Real-Time Database Operations provides the ability to monitor composite operations automatically. The database automatically monitors parallel queries, DML, and DDL statements as soon as execution begins.

Oracle Enterprise Manager Cloud Control (Cloud Control) provides easy-to-use SQL monitoring pages. Alternatively, you can monitor SQL-related statistics using the `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR` views. You can use these views with the following views to get more information about executions that you are monitoring:

- `V$ACTIVE_SESSION_HISTORY`
- `V$SESSION`
- `V$SESSION_LONGOPS`
- `V$SQL`
- `V$SQL_PLAN`

See Also:

- "About Monitoring Database Operations"
- *Oracle Database Reference* to learn about the `V$` views

Application Tracing

A **SQL trace file** provides performance information on individual SQL statements: parse counts, physical and logical reads, misses on the library cache, and so on.

Trace files are sometimes useful for diagnosing SQL performance problems. You can enable and disable SQL tracing for a specific session using the `DBMS_MONITOR` or `DBMS_SESSION` packages. Oracle Database implements tracing by generating a trace file for each server process when you enable the tracing mechanism.

Oracle Database provides the following command-line tools for analyzing trace files:

- `TKPROF`

  This utility accepts as input a trace file produced by the SQL Trace facility, and then produces a formatted output file.

- `trcsess`

  This utility consolidates trace output from multiple trace files based on criteria such as session ID, client ID, and service ID. After `trcsess` merges the trace information into a single output file, you can format the output file with `TKPROF`. `trcsess` is useful for consolidating the tracing of a particular session for performance or debugging purposes.

End-to-End Application Tracing simplifies the process of diagnosing performance problems in multitier environments. In these environments, the middle tier routes a request from an end client to different database sessions, making it difficult to track a client across database sessions. End-to-End application tracing uses a client ID to uniquely trace a specific end-client through all tiers to the database.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_MONITOR` and `DBMS_SESSION`

Optimizer Hints

A **hint** is an instruction passed to the optimizer through comments in a SQL statement.

Hints enable you to make decisions normally made automatically by the optimizer. In a test or development environment, hints are useful for testing the performance of a specific access path. For example, you may know that a specific index is more selective for certain queries. In this case, you may use hints to instruct the optimizer to use a better execution plan, as in the following example:

```
SELECT /*+ INDEX (employees emp_department_ix) */
       employee_id, department_id
FROM   employees
WHERE  department_id > 50;
```

Sometimes the database may not use a hint because of typos, invalid arguments, conflicting hints, and hints that are made invalid by transformations. Starting in Oracle Database 19c, you can generate a report about which hints were used or not used during plan generation.

See Also:

- "Influencing the Optimizer with Hints"
- *Oracle Database SQL Language Reference* to learn more about hints

User Interfaces to SQL Tuning Tools

Cloud Control is a system management tool that provides centralized management of a database environment. Cloud Control provides access to most tuning tools.

By combining a graphical console, Oracle Management Servers, Oracle Intelligent Agents, common services, and administrative tools, Cloud Control provides a comprehensive system management platform.

You can access all SQL tuning tools using a command-line interface. For example, the `DBMS_SQLTUNE` package is the command-line interface for SQL Tuning Advisor.

Oracle recommends Cloud Control as the best interface for database administration and tuning. In cases where the command-line interface better illustrates a particular concept or task, this manual uses command-line examples. However, in these cases the tuning tasks include a reference to the principal Cloud Control page associated with the task.

About Automatic Error Mitigation

The database attempts automatic error mitigation for SQL statements that fail with an ORA-00600 error during SQL compilation.

An ORA-00600 is a severe error. It indicates that a process has encountered a low-level, unexpected condition. When a SQL statement fails with this error during the parse phase, automatic error mitigation traps it and attempts to resolve the condition. If a resolution is found, the database generates a SQL patch in order to adjust the SQL execution plan. If this patch enables the parse to complete successfully, then the ORA-00600 error is not raised and no exception is seen by the application.

How Automatic Error Mitigation Works

These series of examples show how automatic error mitigation can transparently fix ORA-00600 errors.

1. Consider the following error condition. The query has failed and raised a fatal exception.

```
SQL> SELECT count(*)
  2  FROM emp1 e1
  3  WHERE ename = (select max(ename) from emp2 e2 where e2.empno =
e1.empno)
  4    AND empno = (select max(empno) from emp2 e2 where e2.empno =
e1.empno)
  5    AND job = (select max(job) from emp2 e2 where e2.empno = e1.empno);

ERROR at line 3:
ORA-00600: internal error code, arguments: [kkqctcqincf0], [0], [], [],
[], [], [], [], [], [], [], []
```

2. Automatic error mitigation is then turned on in the session.

```
SQL> alter session set sql_error_mitigation = 'on';

Session altered.
```

3. If automatic error mitigation is enabled and it successfully resolves the error, the ORA-00600 message in Step 1 is not displayed, because no exception is returned to the application. The query has been executed successfully.

```
SQL> SELECT count(*)
  2  FROM emp1 e1
  3  WHERE ename = (select max(ename) from emp2 e2 where e2.empno =
e1.empno)
  4    AND empno = (select max(empno) from emp2 e2 where e2.empno =
e1.empno)
  5    AND job = (select max(job) from emp2 e2 where e2.empno = e1.empno);

COUNT(*)

----------
```

```
999
```

4. If you now look at the explain plan for this query, you can see in the **Note** section at the bottom that a SQL patch has been created to repair the query.

```
SQL> SELECT * FROM 
table(dbms_xplan.display_cursor(sql_id=>'5426r24y45gz0',cursor_child_no=>1,
format=>'basic +note'));

PLAN_TABLE_OUTPUT

---------------------------------------------------------------------------
-----
EXPLAINED SQL 
STATEMENT:
------------------------

SELECT count(*) FROM emp1 e1 where ename = (select max(ename) FROM emp2 e2 
WHERE e2.empno = e1.empno) AND empno = (select max(empno) FROM 
emp2 e2 WHERE e2.empno = e1.empno) AND job = (select max(job) FROM emp2 e2 
WHERE e2.empno = e1.empno);


Plan hash value: 
1226419153

-------------------------------------------

| Id  | Operation               | Name    
|
-------------------------------------------

|   0 | SELECT STATEMENT        |         
|
|   1 |  SORT AGGREGATE         |         
|
|   2 |   HASH JOIN             |         
|
|   3 |    HASH JOIN            |         
|
|   4 |     HASH JOIN           |         
|
|   5 |      TABLE ACCESS FULL  | EMP1    
|
|   6 |      VIEW               | VW_SQ_3 
|
|   7 |       SORT GROUP BY     |         
|
|   8 |         TABLE ACCESS FULL| EMP2    
|
|   9 |     VIEW                | VW_SQ_2 
|
|  10 |      SORT GROUP BY      |         
|
|  11 |       TABLE ACCESS FULL | EMP2
```

```
|
|  12 |    VIEW                 | VW_SQ_1
|
|  13 |     SORT GROUP BY       |
|
|  14 |      TABLE ACCESS FULL  | EMP2
|
-------------------------------------------


Note

-----

   - cpu costing is off (consider enabling
it)
   - SQL patch "SYS_SQLPTCH_AUTO_dq7z4ydz3b2ug" used for this statement
```

Tip:

You can get more information about the origin and type of a patched SQL problem by querying `DBA_SQL_PATCHES` and `DBA_SQL_ERROR_MITIGATIONS`.

```
SQL> SELECT name,signature,origin FROM dba_sql_patches
  2  /

    NAME                                SIGNATURE             ORIGIN

----------------------------------------------------------------------
------
    SYS_SQLPTCH_AUTO_dq7z4ydz3b2ug  15789590553029872463  AUTO-
FOREGROUND-REPAIR

SQL> SELECT m.sql_id, m.signature, m.problem_key, m.problem_type 2
FROM dba_sql_error_mitigations m;

    SQL_ID              SIGNATURE
PROBLEM_KEY              PROBLEM_TYPE

----------------------------------------------------------------------
--------------
    5426r24y45gz0      15789590553029872463    ORA 600
[kkqctcqincf0]  COMPILATION ERROR
```

See Also:

Although automatic error mitigation repairs ORA-00600 are transparent to your application, there are views you can inspect to get more information about the process.

The Database Error Message Reference defines ORA-00600, which is the internal error number for Oracle program exceptions.

The *Oracle Database Reference Manual* provides three views related to automatic error mitigation.

- SQL_ERROR_MITIGATION describes the properties of the `SQL_ERROR_MITIGATION` initialization parameter.
- DBA_SQL_ERROR_MITIGATIONS shows the actions performed by automatic error mitigation. It describes each successful error mitigation, based on SQL ID. The `MITIGATION_DETAILS` column provides information on SQL patches created by automatic error mitigation.
- DBA_SQL_PATCHES shows details of SQL patches that have been generated (including but not limited to patches created by automatic error mitigation). The `ORIGIN` column value for patches created by automatic error mitigation is `AUTO-FOREGROUND-REPAIR`.

The *Application Packaging and Types Reference* documents the `SQL_ERROR_MITIGATION` initialization parameter.