Contents

Instructor Preface

1 Following a Tuning Methodology
  Objectives 1-2
  Overview 1-3
  Managing Performance 1-4
  Factors to Be Managed 1-6
  Performance Problems 1-8
  Critical Resource 1-10
  Excessive Demand 1-11
  Understanding Scalability 1-13
  Internet Scalability 1-15
  Scalability with Application Design, Implementation, and Configuration 1-16
  The Tuning Methodology 1-17
  Tuning Roles 1-18
  Tuning SQL Statements 1-19
  Applying the Methodology 1-21
  Summary 1-23

2 SQL Statement Processing
  Objectives 2-2
  Overview 2-3
  System Global Area (SGA) 2-4
  The Shared Pool 2-5
  Shared SQL Areas 2-6
  The Program Global Area (PGA) 2-7
  SQL Statement Processing Phases 2-8
  Sharing Cursors: Benefits 2-12
  Sharing Cursors: Requirements 2-13
  Writing SQL to Share Cursors 2-14
  Bind Variables and Shared Cursors 2-15
  Writing SQL to Share Cursors 2-16
  Monitoring Shared Cursors 2-18
  V$LIBRARYCACHE Columns 2-19
  V$SQLAREA Columns 2-20
  Monitoring Shared Cursor Use 2-21
  Cursor Sharing 2-23
  Automated SQL Execution Memory Management 2-24
  Dynamic Performance Views 2-25
  V$SYSSTAT and V$SESSTAT 2-26
  Summary 2-27
  Practice Overview 2-29
3 EXPLAIN and AUTOTRACE
   Objectives 3-2
   Creating the Plan Table 3-3
   The EXPLAIN PLAN Command 3-5
   EXPLAIN PLAN Example 3-6
   Displaying the Execution Plan 3-7
   Interpreting the Execution Plan 3-9
   Using V$SQL_PLAN 3-11
   V$SQL_PLAN Columns 3-12
   Querying V$SQL_PLAN 3-14
   SQL*Plus AUTOTRACE 3-15
   SQL*Plus AUTOTRACE Examples 3-16
   SQL*Plus AUTOTRACE Statistics 3-17
   Summary 3-18
   Practice Overview 3-19

4 SQL Trace and TKPROF
   Objectives 4-2
   SQL Trace Facility 4-3
   How to Use the SQL Trace Facility 4-4
   Initialization Parameters 4-5
   Switching On SQL Trace 4-7
   Finding Your Trace Files 4-8
   Formatting Your Trace Files 4-9
   TKPROF Command Options 4-11
   Output of the TKPROF Command 4-12
   TKPROF Output Example: No Index 4-17
   TKPROF Output Example: Unique Index 4-18
   Some TKPROF Interpretation Pitfalls 4-19
   Summary 4-20
   Practice Overview 4-21

5 Rule-Based Optimization Versus Cost-Based Optimization
   Objectives 5-2
   Overview 5-3
   Functions of the Oracle9i Optimizer 5-4
   Rule-Based Optimization 5-5
   Cost-Based Optimization 5-6
   Choosing Between RBO and CBO 5-8
   Setting the Optimizer Approach 5-9
   First Rows Optimization 5-10
   Rule-Based Optimization 5-11
   RBO Ranking Scheme 5-12
   Rule-Based Optimization Example 5-13
   Influencing Rule-Based Optimization 5-14
6 Indexes and Basic Access Methods
- Objectives 6-2
- ROWIDs 6-3
- Indexes 6-4
- B*-Tree Indexes 6-5
- B*-Tree Index Structure 6-6
- B*-Tree Index Example 6-7
- CREATE INDEX Syntax 6-8
- Composite Index Guidelines 6-9
- Index Statistics 6-10
- Effect of DML Operations on Indexes 6-11
- Indexes and Constraints 6-12
- Indexes and Foreign Keys 6-13
- Basic Access Methods 6-14
- Skip Scanning of Indexes 6-15
- Identifying Unused Indexes 6-18
- Enabling and Disabling the Monitoring of Index Usage 6-19
- Clusters 6-20
- Cluster Example 6-21
- Index Clusters 6-22
- Index Clusters: Performance Characteristics 6-23
- Index Clusters: Limitations and Guidelines 6-24
- Hash Clusters 6-25
- Hash Clusters: Limitations 6-26
- When to Use Clusters 6-27
- Summary 6-29

7 Collecting Statistics
- Objectives 7-2
- The `ANALYZE` Command 7-3
- Table Statistics 7-5
- Index Statistics 7-7
- Column Statistics 7-10
- The `DBMS_STATS` Package 7-12
- `DBMS_STATS`: Generating Statistics 7-13
- Copy Statistics Between Databases 7-14
- Example: Copying Statistics 7-15
- Example: Gathering Statistics 7-16
- Predicate Selectivity 7-17
- Bind Variables and Predicate Selectivity 7-18
- Histograms 7-20
8 Influencing the Optimizer
Objectives 8-2
Setting the Optimizer Mode 8-3
Some Additional Parameters 8-4
Optimizer Hint Syntax 8-6
Rules for Hints 8-7
Hint Recommendations 8-8
Optimizer Hint Example 8-9
Hint Categories 8-10
Basic Access Path Hints 8-11
Advanced Access Path Hints 8-13
Buffer Cache Hints 8-14
Hints and Views 8-15
View Processing Hints 8-17
Summary 8-18
Practice Overview 8-19

9 Sorting and Joining
Objectives 9-2
Tuning Sort Performance 9-3
Top-N SQL 9-4
Join Terminology 9-5
Join Operations 9-7
Nested Loops Joins 9-8
Nested Loops Join Plan 9-9
Sort/Merge Joins 9-10
Sort/Merge Join Plan 9-11
Hash Joins 9-12
Hash Join Plan 9-13
Joining Multiple Tables 9-14
Outer Joins 9-15
SQL: 1999 Outer Joins 9-16
Full Outer Joins 9-17
Execution of Outer Joins 9-18
The Optimizer and Joins 9-19
Join Order Rules 9-20
How This Book Is Organized

Unit 1: Introduction

<table>
<thead>
<tr>
<th>Lesson</th>
<th>Aim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 1: Following a Tuning Methodology</td>
<td>This lesson covers the process of SQL statement tuning including defining a tuning methodology. By applying a tuning methodology, students learn to optimize performance, improve response time and scalability. Tuning is an ongoing process in the life cycle of a project that needs to start in the development phase and continue through the maintenance phase.</td>
</tr>
<tr>
<td>Lesson 2: SQL Statement Processing</td>
<td>Students learn the four main stages in processing a SQL statement: Parse, Bind, Execute, and Fetch. The advantages and disadvantages of using bind variables are discussed.</td>
</tr>
</tbody>
</table>

Unit 2: Diagnostic Tools

<table>
<thead>
<tr>
<th>Lesson</th>
<th>Aim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 3: EXPLAIN and AUTOTRACE</td>
<td>This unit shows students the tools which are available to help test and diagnose performance problems with a SQL statement; this lesson shows the Explain command and the SQL*Plus AUTOTRACE feature.</td>
</tr>
<tr>
<td>Lesson 4: SQL Trace and TKPROF</td>
<td>This lesson shows the SQL Trace facility. By using SQL Trace, students can evaluate the performance of SQL statements. Trace files can be formatted with TKPROF.</td>
</tr>
</tbody>
</table>

Unit 3: The Oracle Optimizer

<table>
<thead>
<tr>
<th>Lesson</th>
<th>Aim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 5: Rule-Based Optimization Versus Cost-Based Optimization</td>
<td>Students learn to use both rule-based and cost-based optimization. Rule-based optimization is still supported for backward compatibility and uses a ranking scheme. Cost-based optimization was introduced in Oracle7. The optimizer uses statistics to calculate the cost of access paths, taking into account the number of logical I/Os, CPU usage, and network traffic.</td>
</tr>
<tr>
<td>Lesson 6: Indexes and Basic Access Methods</td>
<td>This lesson introduces students to B*-tree indexes. An index is a database object that is logically and physically independent of the table data. The Oracle9i Server can use an index to access data required by a SQL statement, or use indexes to enforce integrity constraints.</td>
</tr>
</tbody>
</table>
Unit 3: The Oracle Optimizer (continued)

<table>
<thead>
<tr>
<th>Lesson</th>
<th>Aim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 7: Collecting Statistics</td>
<td>This lesson covers table statistics used with the cost-based optimizer. By using the <code>ANALYZE</code> command or <code>DBMS_STATS</code> package, students learn that they can collect statistics on tables which are used by the cost-based optimizer to determine optimal execution plans for SQL statements. Histograms are covered too.</td>
</tr>
<tr>
<td>Lesson 8: Influencing the Optimizer</td>
<td>This lesson covers how to influence the Oracle optimizer at three levels: the instance level, the session level, and the statement level (using hints).</td>
</tr>
<tr>
<td>Lesson 9: Sorting and Joining</td>
<td>This lesson covers Top-N SQL, performance issues with sorting, and how to optimize join statements.</td>
</tr>
<tr>
<td>Lesson 10: Optimizer Plan Stability</td>
<td>This lesson covers the use of stored outlines and its influence on optimizer plan stability, and the use of the <code>OUTLN_PKG</code> package on stored outlines.</td>
</tr>
</tbody>
</table>

Unit 4: Advanced Data Access and Storage

<table>
<thead>
<tr>
<th>Lesson</th>
<th>Aim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lesson 11: Advanced Indexes</td>
<td>Students learn about bitmapped indexes, star transformations, and function-based indexes.</td>
</tr>
<tr>
<td>Lesson 12: Materialized Views and Temporary Tables</td>
<td>This lesson shows students materialized views, query rewrites, dimensions and hierarchies, and temporary tables.</td>
</tr>
<tr>
<td>Lesson 13: Alternative Storage Techniques</td>
<td>This lesson shows some alternative storage techniques including index-organized tables, index clusters, and hash clusters.</td>
</tr>
<tr>
<td>Lesson 14: Data Warehousing Considerations</td>
<td>This lesson reviews the efficiencies introduced with the Oracle9i Server that enhance performance in data warehousing environments. The commands <code>WITH</code>, multitable <code>INSERT</code>, and <code>MERGE</code> statements are reviewed from a performance perspective.</td>
</tr>
</tbody>
</table>
Following a Tuning Methodology

**Schedule:**

<table>
<thead>
<tr>
<th>Timing</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 minutes</td>
<td>Lecture</td>
</tr>
<tr>
<td>40 minutes</td>
<td>Total</td>
</tr>
</tbody>
</table>

**Instructor Note**

Make sure to mention the suggested prerequisite for this course: *Oracle9i DBA Fundamentals I*. Do not offer too much detail about indexing and clustering techniques; just use them, and measure performance. That is the main objective of this course. Also, the main target audience for this course is developers, not database administrators.

Do not explain and demonstrate everything while teaching the lessons; the students should discover certain things while participating in the workshops.

The first lesson provides participants with a framework for the course. It explains why tuning is necessary, putting tuning firmly in the realm of solving business problems. It also presents a tuning methodology and shows how the topic of this course fits into that methodology.
Objectives

After completing this lesson, you should be able to do the following:

• State the procedural steps in managing performance
• Describe the causes of performance problems
• Identify the main system areas that you can address by using the tuning process
• Describe the tuning methodology
• Explain the advantage of following the tuning methodology in its proper order
• List the tuning steps that are the responsibility of the application developer

Objectives

Applying a tuning methodology to your design and tuning efforts optimizes performance. Tuning SQL is a major component of a tuning methodology.

After completing this lesson, you should be able to:

• State the procedural steps in managing performance
• Describe the causes of performance problems
• Identify the main system areas that you can address by using the tuning process
• Describe the tuning methodology
• Explain the advantage of following the tuning methodology in its proper order
• List the tuning steps that are the responsibility of the application developer
Overview

Anyone involved in tuning should follow a tuning methodology to achieve maximum performance. Tuning SQL statements is an important step that is least expensive when performed at its proper moment within the methodology.

In addition to tuning at the right time, you should also have a good understanding of the issues involved in performance management and the types of performance problems that might arise.
Managing Performance

Tuning requires several steps.

Start Early

Performance management must span the application or project continuously to be fully effective. You should consider it at the design stage.

Set Objectives

Do not attempt to tune everything. Use the ROI (return on investment) methodology to identify the portions of the application or project to tune. Define your objectives in a form accepted and agreed to by all the interested parties. Service level agreements (SLAs) form an increasingly important part of setting the objectives of operations groups and facilities management teams. You can successfully extend this concept into the specification of application requirements.

It is better to set a measurable objective such as “this report will complete in 5 seconds or less” than to aim for “all queries will be faster.”

Tune and Monitor Conformance

After you have set out and agreed upon the objectives, you are ready to tune and to reach those objectives, monitoring your progress as you tune. You should keep detailed records about the level of conformance with the requirements. You should publish indicative measures at regular intervals, highlighting any deviations or trends.
Managing Performance (continued)

Work Together
Database administrators, system administrators, and programmers should work together as a team, not as adversaries.

Handle Exceptions and Changes
When effective monitoring is in place and supported at the same level as the rest of the application, you must react promptly to any changes when they are reported. Use the available data to select a course of action and then analyze the resulting performance to determine whether your actions are successful. If overall changes in use patterns or equipment capability occur, then you should consider setting new objectives.

The 80/20 Rule
Typically, 20% of your transactions (including select statements) account for 80% of the system usage. The transactions that make up this 20% are the ones that must be tuned, and tuning anything else is probably not worth your time and energy.
You can also formulate this phenomenon with the following corollary: 50% of the utilization comes from 5% of your transactions.
Factors to Be Managed

- Schema
  - Data design
  - Indexes
- Application
  - SQL statements
  - Procedural code
- Instance
- Database
- User expectations

Instructor Note

Point out to the participants that:
- Schema design, as covered in Oracle9i Database Fundamentals for Developers, is a suggested prerequisite for this course.
- Tuning SQL statements is the focus of this course.
- Tuning procedural code (PL/SQL, 3GLs) is part of learning that particular coding environment.
- Instance and database tuning are discussed in the Oracle9i Performance Tuning course. Refer to them briefly, when necessary.
Factors to Be Managed

Performance management can be divided into the following four areas. Although the areas are separate, they are also interdependent and require different skill sets.

- Schema tuning deals with the physical structure of the data.
- Application tuning deals with the business functions and the program modules that implement the functions. Tuning the procedural code for the type of application and tuning the embedded SQL statements are also included in this area.
- Instance tuning deals with the installation of the Oracle9i Server and how it uses memory.
- Database tuning deals with managing the physical arrangement of data on the disk.

This course concentrates on schema and SQL statement tuning. Other parts of application tuning should be part of learning that particular type of application. Instance and database tuning are covered in courses and literature for database administrators.

Schema

If an application has inadequate or inappropriate data design, then tuning the physical allocation, providing indexes, or rewriting programs will not overcome the problem.

SQL Statements

If an application is well designed, it may still perform badly. A common reason for this is badly written SQL.

User Expectations

Users expect consistent performance. They can cope with slower application functions if they understand why the application is slower than usual. The project team should try to build a realistic user expectation of performance, possibly including application messages to warn operators that they are requesting operations that are resource-hungry. The best time to do this is before the design and build phases and as part of the transition phase.
Performance Problems

• Inadequate consumable resources
  – CPU
  – I/O
  – Memory (may be detected as an I/O problem)
  – Data communications resources
• Design limitations
• Locking

Instructor Note

Before turning to the next page, you could ask for suggestions about performance dependencies and write them on the board.
Performance Problems

Performance problems occur when a function takes longer to perform than the time allowed. This is because a resource of a particular type is insufficient or inadequate. The resource may be a physical resource, such as available memory buffers that store data blocks, or an artificial resource, such as a lock.

Inadequate Consumable Resources

A resource may simply be inadequate to meet the need under any circumstances. For example, if you want a function to complete in under one second, a network with a message turnaround time of two seconds will never meet the target.

If the limiting factor is a consumable resource, such as CPU power, all the users of that resource are affected.

Inadequate Design Resources

If the limiting factor is the contention of processes for a design resource, such as a lock, then only users of those specific processes are likely to be affected.

Locking

Contention due to locking by other transactions or applications might be a problem.
Critical Resource

- Performance depends on the following:
  - How many clients need the resource
  - How long they must wait for it
  - How long they hold it
- Consider limiting demand to maintain acceptable response times.

Response time is defined as the *service time* plus the *wait time* to accomplish a certain task. As demand for a resource with a single server increases toward the service rate, queues build up with the queue length, increasing exponentially for each increase in demand.

Even if there are many servers, you can observe the same effect with a single queue. However, multiple servers provide a valuable smoothing effect when there is a wide variation in the time for which a resource is occupied by one of its clients before it is available for reallocation.

The goal is to design, engineer, and tune the system so that the load is never permitted to slow service completion times below the acceptable level.
Excessive Demand

- Greatly increases response time and reduces throughput
- Should be prevented as much as possible by limiting demand to a level that still allows reasonable throughput

Excessive Demand

Throughput is defined as the total amount of work accomplished by the system in a given amount of time. Too many processes using a system simultaneously may result in the following symptoms.

Increased Response Time

Most users know and understand the effects of queues on increasing response time. They may be prepared to accept slower response at peak times if the effect is linear. However, both statistical theory and experience show that when response time starts to deteriorate, small increases in load can have a severe effect, which is unacceptable to users.
Excessive Demand (continued)

Reduced Throughput

Any marked degradation in response time is likely to disrupt the work rate of the users who are affected. Many people do not understand that adding more users to a system significantly decreases the overall throughput of the system.

If system throughput is important, you must ensure that the number of users does not exceed the threshold at which the throughput starts to decline. It is best to limit the number of users mechanically.

This may force a change in working patterns to cope with the restriction, but spreading the peak periods across the working day can improve the way the system is used.
Understanding Scalability

- A system’s ability to process more workload, with a proportional increase in system resource usage
- Aimed at application designers and performance specialists
- Poor scalability leads to system resource exhaustion to the point where no more throughput is possible when its workload is increased.

Scalability

Scalability is a system’s ability to process more workload, with a proportional increase in system resource usage. In other words, in a scalable system, if you double the workload, then the system must use twice as many system resources.

Examples of bad scalability due to resource conflicts include the following:
- Applications requiring significant concurrency management as user populations increase
- Increased locking activities
- Increased data consistency workload
- Increased operating system workload
- Transactions requiring increases in data access as data volumes increase
- Poor SQL and index design resulting in a higher number of logical I/Os for the same number of rows returned
- Reduced availability because database objects take longer to maintain
Understanding Scalability

Resource exhaustion due to:

- Hardware exhaustion
- Table scans in high-volume transactions, causing inevitable disk I/O shortages
- Excessive network requests, resulting in network and scheduling bottlenecks
- Memory allocation, causing paging and swapping
- Excessive process and thread allocation, causing operating system thrashing

Understanding Scalability

If an application exhausts a system resource to the point where no more throughput is possible when its workload is increased it is said to be unscalable. This can result in fixed throughputs and poor response times.
Internet Scalability

Characteristics of Internet applications:
- Availability 24 hours a day, 365 days a year
- An unpredictable and imprecise number of concurrent users
- Difficulty in capacity planning
- Availability for any type of query
- Multitier architectures
- Stateless middleware
- Rapid development timescale
- Minimal time for testing

Internet Scalability

Applications must scale with the increase in workload and also when additional hardware is added to support increasing demand. Design errors can cause the implementation to reach its maximum, regardless of additional hardware resources or redesign efforts.

Internet applications are challenged by very short development timeframes with limited time for testing and evaluation. A badly designed application generally means that at some point in the future, the system will need to be rearchitected or reimplemented. From a business perspective, poor performance can mean a loss of customers. If a Web user does not get a response in seven seconds, then the user’s attention could be lost forever.

In many cases, the cost of redesigning a system with the associated down time costs in migrating to new implementations exceeds the costs of properly building the original system. It is critical to design and implement with scalability in mind from the start.
Scalability with Application Design, Implementation, and Configuration

Applications have a large impact on scalability.
- Poor schema design can cause expensive SQL that does not scale.
- Poor transaction design can cause locking and serialization problems.
- Poor connection management can cause poor response times and unreliable systems.

However, the design is not the only problem. The physical implementation of the application can be the weak link. For example:
- Systems can move to production environments with bad I/O strategies.
- The production environment could use different execution plans than those generated in testing.
- Memory-intensive applications that allocate a large amount of memory without much thought for freeing the memory at run time can cause excessive memory usage.
- Inefficient memory usage and memory leaks put a high stress on the operating virtual memory subsystem. This impacts performance and availability.
The Tuning Methodology

1. Tune business function
2. Tune data design
3. Tune process design
4. Tune SQL statements
5. Tune physical structure
6. Tune memory allocation
7. Tune I/O
8. Tune memory contention
9. Tune operating system

The list in the slide shows the phases of the development cycle in which tuning can be applied. Following the steps in this order is recommended for the following reasons:

- The earlier the step, the higher the potential for improving performance.
- The earlier the step, the higher the cost for performing or reworking it later. For example, changes to the data structures and application code after the initial design tend to be expensive and require the additional expense of retesting the affected components.
- Decisions made at one step can influence subsequent steps. For example, the SQL statement that you write in Step 4 will have significant bearing on parsing and caching issues that are addressed in Step 6.

The more extensively that you use object-oriented design techniques and multitiered architecture, the better the chances that you will safely achieve any application changes at a reasonable cost.

Instructor Note

About object-oriented design techniques: on the other hand, encapsulation tends to make you lose control over events occurring at a lower level.
Tuning Roles

The business analyst, designer, application developer, database administrator, and operating system administrator are responsible for different steps in the tuning process. In some cases, one person may fulfill several of these roles.

The steps performed by the database administrator and operating system administrator have a lesser effect on performance than earlier steps, but they can be performed at relatively low cost with immediately available and observable results.

The fourth step in the methodology is primarily the responsibility of the application developer. However, understanding how to tune SQL can enable designers to design schemas that tune easily. With this understanding, database administrators can help pinpoint SQL tuning needs and solutions, thus easing the burden on their databases. This is especially useful if the application is already in production and the application developer is no longer available.

A large part of tuning in client-server or Web environments relates to network issues, as well.
Tuning SQL Statements

Use performance analysis tools to check the results of the following steps:

1. Tune the schema.
   - Add indexes as required
   - Consider index-organized tables vs. heap-organized tables in certain situations
   - Create clusters if needed

2. Choose the language: SQL, PL/SQL, or Java.

Tuning SQL Statements

During the steps within SQL statement tuning, analysis techniques should be used frequently to determine goals and progress.

1. Tuning the Schema
   The designer is responsible for determining which tables should be used, their contents, and similar issues. The application developer may then need to decide when to use indexing, when to use index-organized tables, and when to use clustering. Denormalization may be needed for good performance, especially in data warehouse environments. The decisions that are made at this step greatly affect the performance of any SQL statements that use these structures.

2. Choosing a Language: SQL, PL/SQL, or Java
   In some cases, you can achieve better performance by using the PL/SQL or Java language to perform a task.
Tuning SQL Statements

3. Design for the reuse of SQL optimization.
4. Design and tune SQL statements.
5. Maximize performance with the optimizer.

Tuning SQL Statements (continued)

3. Designing for Reuse of SQL Parsing, Optimization, and Tuning Efforts
   The Oracle9i Server is often able to reuse some of its efforts at parsing and optimization when it sees an identical statement repeated. Therefore, creating SQL statements that appear identical can improve performance. It also enables the application developer to focus and tune individual SQL statements that are used repeatedly.

4. Designing and Tuning SQL Statements
   There is a wide variety of methods for designing high-performance SQL statements. Knowing the overall functions of the optimizer helps to show where tuning efforts can be effective. Consider rewriting SQL statements into semantically equivalent statements. Also, use the correct language (SQL or PL/SQL) in each situation.

5. Achieving Maximum Performance with the Optimizer
   To make the best use of the Oracle cost-based optimizer, you must understand how it chooses access paths to data. You must assist the cost-based optimizer by using the ANALYZE command and by sometimes providing it hints about the best access path. The rule-based optimizer is also available. Both the cost-based optimizer and the rule-based optimizer will be discussed in the “Rule-Based Optimization Versus Cost-Based Optimization” lesson.
Applying the Methodology

• Be proactive: work from the top down.
• If you must be reactive, work from the bottom up, using these hints:
  – Establish quantifiable objectives.
  – Create a minimum repeatable test.
  – Ask questions of affected users, and avoid preconceptions.
  – Test hypotheses and keep notes.
  – Stop when you meet the target.

Quantifiable Objectives
Establish quantifiable objectives and avoid vague ones. An example of an acceptable objective would be the following:

“We must be able to support up to 20 operators, each entering 20 orders per hour, and the picking lists must be produced within 30 minutes of the end of the shift.”

Minimum Repeatable Test
If you are trying to cut a four-hour run down to two hours, repeated timings take too long. Perform your initial trials against a test environment similar to the real one (or provide input for a restrictive condition, such as processing one department instead of 500 of them). The ideal test case should run for more than one minute so that improvements are demonstrated intuitively and can be measured using timing features.

• Ask questions of affected users and avoid preconceptions.
• Test hypotheses and keep notes.
• Stop when you meet the target.
Tuning a Specific Problem for an Application Already in Production

Often, performance specialists are summoned late in the life cycle of a project, when the system is in production and performing unacceptably.

In this situation, start at the bottom of the method list and work up. The items at the end are the cheapest and fastest to perform. If they do not solve the problem, you will have to work back “uphill,” and this starts to increase costs and time delays.
Summary

In this lesson, you should have learned how to:

- **Manage performance**
  - Start early: be proactive
  - Set measurable objectives
  - Monitor requirements conformance
  - Handle exceptions and changes
- **Identify performance problems**
  - Inadequate consumable resources
  - Inadequate design resources
  - Critical resources
  - Excessive demand
Summary

• Tune SQL statements
  – Manage user expectations
  – Analyze the results at each step
  – Tune the physical schema
  – Choose when to use SQL
  – Reuse SQL statements when possible
  – Design and tune the SQL statement
  – Get maximum performance with the optimizer

Summary (continued)

Tuning SQL Statements

Tuning SQL statements involves using analysis tools, SQL tuning techniques, and the optimizer effectively. In addition, tuning the schema can create additional access paths for the optimizer, such as the use of an index.
SQL Statement Processing

Schedule: Timing Topic
60 minutes  Lecture
15 minutes  Practice
75 minutes  Total

Instructor Note
This lesson provides insight into the way SQL statements are processed by the Oracle9i Server. It explains the Parse, Bind, Execute, and Fetch phases, concentrating on minimizing parsing and optimizing shared cursor usage. It also discusses bind variables in that perspective.
Objectives

After completing this lesson, you should be able to do the following:

• Describe the basic steps involved in processing a SQL statement
• Monitor the use of shared SQL areas
• Write SQL statements to take advantage of shared SQL areas
• Understand how to use the CURSOR_SHARING parameter
• Use automatic PGA memory management

Knowing how the Oracle9i Server processes SQL statements can help you design SQL statements that can reuse parsing and optimization efforts.

After completing this lesson, you should be able to:

• Describe the basic steps involved in processing a SQL statement
• Monitor the use of shared SQL areas
• Write a SQL statement to take advantage of shared SQL areas (This will enable you to reuse parsing and optimization efforts).
• Understand how and when to set the CURSOR_SHARING parameter to SIMILAR or FORCE.
• Use the automatic PGA memory management.
Overview

By knowing about shared SQL areas, the SQL processing phases, and shared cursors, you can understand how coding standards enable you to minimize how often your statements must be parsed and optimized.

Additionally, as the statements you write become more and more standardized, you can identify statements that occur frequently and devote additional tuning to them.
System Global Area (SGA)

The System Global Area (SGA) is a memory area used to store database information that is shared by database processes. It is allocated in the virtual memory of the computer where the Oracle server resides. The SGA contains several subareas that are used to organize and hold different types of information. The Oracle server uses memory to store various types of information, such as program code being executed, information about connected sessions, SQL statements, and so on. One area is the program global area (PGA), another is the shared pool. Various processes manage this area of memory.

Instructor Note (for page 2-5)

A parse tree is a structure in memory where each node on the tree represents processing instructions for a step in the execution plan. In executing the plan, the processor follows these nodes sometimes recursively.
The Shared Pool

Contains:

- Data dictionary cache
- Library cache
  - SQL statements
  - Parsed or compiled PL/SQL blocks
  - Java classes

The shared pool is a portion of the system global area that contains shared memory constructs such as shared SQL areas, the data dictionary cache, and the fully parsed or compiled representations of PL/SQL blocks.

The main components of the shared pool are the library cache and the dictionary cache. The library cache stores the executable (parsed or compiled) form of recently referenced SQL and PL/SQL code. The dictionary cache stores information such as usernames, segment information, tablespace information, and sequence numbers.

- A shared SQL area is required to process every unique SQL statement submitted to a database.
- A shared SQL area contains information such as the parse tree and execution plan for the corresponding statement.
- A single shared SQL area is used by multiple applications that issue the same statement, and therefore needs fewer hard parses which results in performance improvement as well as leaving more shared memory for other uses.

The database administrator (DBA) can change the available dictionary and shared SQL areas space by modifying the `SHARED_POOL_SIZE` initialization parameter. The DBA does this as part of an overall database tuning effort, which is subject to various constraints discussed in the Oracle9i Performance Tuning course.

Oracle9i: SQL Tuning Workshop 2-5
Shared SQL Areas

When you invoke application code, Oracle attempts to reuse existing code if it has been executed previously so that it can be shared. If the parsed representation of the statement does exist in the library cache and it can be shared, Oracle reuses the existing executable.

When the shared pool in the SGA fills up, the Oracle server makes room for newly requested programs by using a least recently used (LRU) algorithm, so that items in the shared pool that have not been used often are aged out to make room for the new programs.

**Cursors**

Inside the shared SQL area, each SQL statement is parsed in its own part, known as a context area or cursor.

Each cursor holds the following information:

- The parsed statement (static, dynamic, and recursive SQL, plus program units such as procedures and database triggers)
- The execution plan
- A list of referenced objects

If two users issue the same SQL statement, then they will use the same cursor. The statement is reparsed if the representation in the shared pool is invalid. This happens, for example, if a data definition language (DDL) statement such as `ALTER TABLE` was used on one of the objects in the statement, or if a dependent table is analyzed.
The Program Global Area (PGA)

- Is a memory buffer that contains data and control information for a server process.
- Can control and tune the size of a work area.
  - Optimal size
  - One-pass size
  - Multipass size

PGA

A program global area (PGA) is a memory region containing data and control information for a single process (server or background). Access to it is exclusive to that server process and is read and written only by the Oracle code acting on behalf of it. PGA is a nonshared memory area to which a process can write. One PGA is allocated for each server process. A PGA is allocated by the Oracle server when a user connects to an Oracle database and a session is created, though this varies by operating system and configuration.

An example of such information is the run-time area of a cursor. Each time a cursor is executed, a new run-time area is created for that cursor in the PGA. With complex queries, a big portion of the run-time area is dedicated to work areas allocated by memory-intensive operators such as sort-based operators. A sort operator uses a work area (the sort area) to perform the in-memory sort of a set of rows.

The size of the work area can be controlled and tuned. Ideally, the size of a work area is big enough that it can accommodate the input data and auxiliary memory structures allocated by its associated SQL operator. This is called the optimal size of a work area. When the size of the work area is smaller than optimal, response time increases, because an extra pass is performed over part of the input data. This is called the one-pass size of the work area. Under a one-pass threshold, when the size of a work area is much too small compared to the input data size, multiple passes over the input data are needed. This is called the multipass size of the work area.
Processing Phases

The four most important phases in SQL statement processing are parsing, binding, executing, and fetching.

The reverse arrows indicate processing scenarios; for example, Fetch—(Re)Bind—Execute—Fetch.

The Fetch phase applies only to queries and DML statements with a returning clause.

Note: A detailed description of SQL statement processing can be found in the Oracle9i Application Developers Guide - Fundamentals, “How Oracle Processes SQL Statements” and the Oracle9i Concepts, “SQL and PL/SQL.”

Instructor Note

There is no real Rebind phase; the same pointer is simply used for a re-Execute. The value found at the corresponding memory address is used; Oracle9i uses a “bind by reference” technique.

Three more phases are missing from this slide: Define, Describe, and Parallelize.
Define and Describe are not very important for tuning. Parallelize is not mentioned here because it is discussed in the Oracle9i: Implementing Scalable Systems course.
SQL Statement Processing Phases

• Parse
  – Searches for the statement in the shared pool
  – Checks syntax
  – Checks semantics and privileges
  – Merges view definitions and subqueries
  – Determines execution plan

Parse Phase
The Oracle9i Server does the following:
  • Searches for the statement in the shared pool
  • Checks the statement syntax, given the grammar and specifications of the SQL language
  • Checks the semantics, ensuring that objects referenced in the SQL statement are valid and satisfy security constraints
  • Determines whether the process issuing the statement has appropriate privileges to execute it
  • Transforms a SQL statement on a view into an equivalent SQL statement on its underlying definition, and attempts to simplify a statement with a subquery by rewriting it into a join
  • Determines and stores the execution plan, or uses an existing execution plan, if possible

Note: The Parse phase also checks whether materialized views can be used. Materialized views are discussed in a subsequent lesson.

Instructor Note
Determining the execution plan is the most costly part of the parsing and may take longer than executing the plan.
SQL Statement Processing Phases

- **Bind**
  - Scans the statement for bind variables
  - Assigns (or reassigns) a value

**Bind Phase**
- The Oracle9i Server checks the statement for references of bind variables.
- The Oracle9i Server assigns or reassigns a value to each variable.

**Note:** This phase order implies that the Oracle9i Server does not know bind variable values when optimizing a statement. This enables a fast rebinding-execute without the need for reparsing, thus saving time and memory; a disadvantage is that it is impossible for the optimizer to estimate predicate selectivity. This will be discussed in more detail in the “Collecting Statistics” lesson.

**Instructor Note (for page 2-11)**

Note the subtle difference between sorts for DML statements (during the Execute phase) and queries (during the first fetch). There is no need to stress this, but students might ask.

With an INSERT, UPDATE, or DELETE statement, executing the statement results in the SQL being run immediately during the Execute phase.

When a SELECT statement where rows are sorted or locked is issued, opening the cursor retrieves all of the rows to be returned. If there is no locking or sorting, opening the cursor locates the record pointer at the first row.

*Oracle9i: SQL Tuning Workshop 2-10*
SQL Statement Processing Phases

- Execute
  - Applies the execution plan
  - Performs necessary I/O and sorts for data manipulation language (DML) statements

- Fetch
  - Retrieves rows for a query
  - Sorts for queries when needed
  - Uses an array fetch mechanism

Execute Phase

- The Oracle9i Server applies the parse tree to the data buffers.
- Multiple users can share the same parse tree.
- The Oracle9i Server performs physical reads or logical reads/writes for DML statements and also sorts the data when needed.

Fetch Phase

The Oracle9i Server retrieves rows for a SELECT statement during the Fetch phase. Each fetch typically retrieves multiple rows, using an array fetch.

Each Oracle tool offers its own ways of influencing the array size; in SQL*Plus you do so by using the ARRAYSIZE setting:

```
SQL> show arraysize
arraysize 15
SQL> set arraysize 1
```

With this setting, SQL*Plus will process one row at a time. The default value is 15.
Shared Cursor Benefits

When a SQL statement is found in the shared SQL area, then the Parse phase is curtailed and
the existing cursor is used. These are the main benefits of sharing cursors:

- Sharing cursors reduces parsing and saves time.
- Memory dynamically adjusts to the SQL being executed.
- Memory usage may improve dramatically, even for tools that store SQL within the
  application.
Sharing Cursors: Requirements

Cursors can be shared only by SQL statements that have the following identical elements:

- **Text**
  - Uppercase and lowercase
  - White space (spaces, tabs, carriage returns)
  - Comments
- **Referenced objects**
- **Bind-variable data types**

---

Shared Cursor Requirements

- Only identical SQL statements can use the same cursor.
- The text of the SQL statements must be exactly the same, including case, spaces, tabs, carriage returns, and comments.
- The objects referenced in the SQL statements must resolve to the same objects in the database.
- The types of the bind variables used in the SQL statements must be the same.

**Note:** Before sending SQL statements to the Oracle9i Server, most Oracle tools (such as PL/SQL, the precompilers, and Oracle Developer) preprocess SQL statements to make them as identical as possible by removing comments, squeezing white space, and converting to uppercase or lowercase.

SQL*Plus, however, sends SQL statements to the Oracle9i Server in the same format as they are entered.

**Instructor Note (for page 2-14)**

Monitoring shared SQL is discussed at the end of this lesson. You can demonstrate these four SQL statements at the end of this lesson—not now.
Writing SQL to Share Cursors

- If the case or the amount of white space is different, then the statements are not identical.

  ```sql
  select * from customers where cust_id = 180;
  select * from CUSTOMERS where CUST_ID = 180;
  ```

- If the objects belong to different users, then the statements are not identical.

  ```sql
  select * from customers where CUST_ID = 180;
  select * from customers where CUST_ID = 180;
  ```

Writing SQL to Share Cursors

SQL statements must be identical to be able to share cursors. Note that sharing statements is unimportant in a decision support system (DSS) environment, because most statements will be different anyway.

Identical Case

The first two examples in the slide are not identical. Notice the case difference for the table and column names. Because of this case difference, the statements are not identical and thus cannot share a SQL area. Because there is case difference between the two statements, the parser gives different internal codes (HASH_VALUES) for them. This can be checked watching the V$SQL view.

Identical Objects

Even when two statements look identical, if the objects actually refer to different database objects, then the two statements are not identical.

In the last two examples in the slide, the statements are issued by two different users who each have their own CUSTOMERS table. Thus the statements are not identical and cannot share a SQL area.
Bind Variables and Shared Cursors

• Suppose you enter the following two statements:

```
select * from customers where cust_id = :c
```
```
select * from customers where cust_id = :d
```

• Both statements are translated into:

```
select * from customers where cust_id = :b1
```

Bind Variables
If two bind variables have different data types, then the statements are not identical. If the bind-variable data types match but their names are not identical, as in the example above, there is no problem, because bind variables will be renamed internally. The first variable is always called :b1, the second is :b2, and so on.

Instructor Note
Demo scripts: demo02_01.sql
demo02_01cursors.sql

Note that the script actually uses the PL/SQL environment of SQL*Plus. SQL*Plus itself does not translate bind-variable references in SQL statements.

Note that starting with version 8.1.6, there is a powerful feature to force cursor sharing, which you can use to set CURSOR_SHARING=FORCE at the session or instance level. This will replace all literals in your SQL statements by system-generated bind variables. This option is discussed at the end of the lesson.
Writing SQL to Share Cursors

Create generic code using the following:
- Stored procedures and packages
- Database triggers
- Referenced Oracle Forms Services procedures
- Any other library routines and procedures

Writing SQL to Share Cursors

Develop coding conventions for SQL statements in ad hoc queries, SQL scripts, and Oracle Call Interface (OCI) calls.

Using Generic Shared Code
- Write and store procedures that can be shared across applications.
- Use database triggers.
- Write referenced triggers and procedures when using Oracle Developer.
- Write library routines and procedures in other environments.
Writing SQL to Share Cursors

Follow SQL coding standards:

- Case
- White space
- Comments
- Object references
- Bind variables

Writing SQL to Share Cursors (continued)

Writing to Format Standards

- Develop format standards for all statements, including those in PL/SQL code.
- Develop rules for use of uppercase and lowercase.
- Develop rules for use of white space (spaces, tabs, carriage returns).
- Develop rules for use of comments, preferably keeping them out of the SQL statements themselves.
- Use the same names to refer to identical database objects.
Monitoring Shared Cursors

- **V$LIBRARYCACHE** provides general information about the library cache.
- Information about individual SQL statements is contained in the following views:
  - **V$SQLTEXT**
  - **V$SQLTEXT_WITH_NEWLINES**
  - **V$SQL_BIND_DATA**
  - **V$SQL** and **V$SQLAREA**

Instructor Note

Querying V$ views is usually a DBA task. Tell the participants that they may not have access to these data dictionary views on their home systems. The following pages give examples of V$LIBRARYCACHE and V$SQLAREA usage only.

Another interesting view is **V$DB_OBJECT_CACHE**.

Note that **V$SQL** lists all copies of identical SQL statements (child cursors) on a separate line, whereas **V$SQLAREA** groups those statements together.

Do not go into too much detail here. Use these pages to show shared SQL. The *Oracle9i Performance Tuning* course covers this in more detail.
The **V$LIBRARYCACHE** View

This view holds information about library cache management. Values for **PINHITRATIO** and **GETHITRATIO** close to 1 indicate a good library cache performance. Note that this slide does not show all columns of the **V$LIBRARYCACHE** view.

The **V$LIBRARYCACHE** view is used in the query below to check on the amount of caching.

```
select gethitratio, pinhitratio
from   v$librarycache
where  namespace = 'SQL AREA';
```

```
GETHITRATIO  PINHITRATIO
-----------  -----------
.968244859   .997191864
```

Low **GETHITRATIO** could indicate that objects are getting swapped out of memory. Low **PINHITRATIO** could indicate that the session is not executing the same cursor multiple times (even though it might be shared across different sessions), or the session is not finding the cursor shared.

**Instructor Note**

Demo script: demo02_02.sql

Instead of the **PINHITRATIO**, you could also use `sum(reloads)/sum(pins)` ratio to determine when to increase the size of the shared pool.

---

**V$LIBRARYCACHE Columns**

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAMESPACE</td>
<td>The name of the library cache area</td>
</tr>
<tr>
<td>GETS</td>
<td>Total number of requests (lookups)</td>
</tr>
<tr>
<td>GETHITS</td>
<td>The number of times that an object’s handle was found in memory</td>
</tr>
<tr>
<td>GETHITRATIO</td>
<td>The ratio of GETHITS to GETS</td>
</tr>
<tr>
<td>PINS</td>
<td>The number of objects in the library cache</td>
</tr>
<tr>
<td>PINHITS</td>
<td>The number of times that all the pieces of the object were found in memory</td>
</tr>
<tr>
<td>PINHITRATIO</td>
<td>The ratio of PINHITS to PINS</td>
</tr>
<tr>
<td>RELOADS</td>
<td>The number of library cache misses</td>
</tr>
</tbody>
</table>

---

Copyright © Oracle Corporation, 2002. All rights reserved.
V$SQLAREA Columns

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_TEXT</td>
<td>Text of the SQL statement</td>
</tr>
<tr>
<td>VERSION_COUNT</td>
<td>Number of versions of this cursor</td>
</tr>
<tr>
<td>LOADS</td>
<td>Number of times the cursor has been loaded</td>
</tr>
<tr>
<td>INVALIDATIONS</td>
<td>Number of times the contents have been invalidated</td>
</tr>
<tr>
<td>PARSE_CALLS</td>
<td>Number of times a user has called this cursor</td>
</tr>
<tr>
<td>SORTS</td>
<td>Number of sorts performed by the statement</td>
</tr>
<tr>
<td>COMMAND_TYPE</td>
<td>Command type</td>
</tr>
<tr>
<td>PARSING_USER_ID</td>
<td>Parsing user ID (SYS = 0)</td>
</tr>
</tbody>
</table>

Shared Cursor Use

The V$SQLAREA view holds information about all shared cursors in the cache.

- **VERSION COUNT > 1**: Indicates that the same text is used by different users on their own version of a table.
- **LOADS > 1**: Indicates cursor reloads after aging out or cursor invalidation.
- **COMMAND_TYPE**
  - 1: CREATE TABLE
  - 2: INSERT
  - 3: SELECT
  - 6: UPDATE
  - 7: DELETE

**Note:** Only the most important columns of the V$SQLAREA view are listed above.

Instructor Note

V$SQL provides more detailed information than V$SQLAREA, because it does not group by the SQL statement text. For a full list of command type values, see description of the V$SESSION view in Oracle9i Server Reference.
Monitoring Shared Cursor Use

- One load per statement is ideal.
- One load per version/invalidation is optimal.
- More than one load per version indicates a potential benefit from increasing the shared pool size.

In the best case scenario, there should be one version of each statement that is never invalidated or aged out.

If the number of loads is significantly higher than the sum of the versions and invalidations, especially if the number of loads is similar to the number of calls, then the cursor has probably been reloaded because of aging, and the system may benefit from increasing the size of the shared pool.
Monitoring Shared Cursor Use

The statement above excludes information about recursive SQL (parsing user SYS: user_id=0) and displays only SELECT commands (command type 3).

There are two versions of the first statement, probably because they reference two different CUSTOMERS objects. However, each version has been loaded only once. The statement has been issued three times (PARSE_CALLS).

There is only one version of the second statement, but it has been loaded twice, having been invalidated once (probably by some DDL on the table or related index).

Note: Oracle SQL Analyze, an add-on component of the Oracle Enterprise Manager (OEM) Tuning Pack, offers an excellent graphical interface on top of V$SQLAREA.

Instructor Note

V$SQL would distinguish between the two versions of the first SQL statement.

Demo scripts: demo02_03.sql
demo02_01cursors.sql

Edit script demo02_03.sql to fill in the correct user ID and location prior to running the script.
Cursor Sharing

Use the parameter 
CURSOR_SHARING=SIMILAR|FORCE when:

- You want to match to previous statements that differ only in literal values.
- Response time is low due to very high library cache misses.

Cursor Sharing

The parsing phase compares the text of the statement with existing statements in the shared pool to see whether the statement can be shared. If the statement differs textually in any way, then Oracle does not share the statement.

The CURSOR_SHARING parameter enables the DBA to specify the level of matching that will be considered acceptable by the SQL parser. The default value is EXACT, which forces the parser to seek a statement that exactly matches the current statement before reusing an existing cursor.

Setting CURSOR_SHARING to SIMILAR allows the parser to use a cursor for a statement that is identical in all aspects other than literal values. After the statement is identified, the current statement will continue to be parsed to ensure that the cursor’s execution plan is applicable to the current statement. CURSOR_SHARING = FORCE makes the parser behave the same way as for SIMILAR, except the execution plan in the cursor is always used, regardless of its applicability.

CURSOR_SHARING = SIMILAR or FORCE is not recommended for certain environments such as DSS or for complex queries.

See the Oracle9i Database Performance Guide and Reference for more information.
Automated SQL Execution Memory Management

- Allocation and tuning of PGA memory is simplified and improved.
- DBAs can use parameters to specify the policy for PGA sizing.
Dynamic Performance Views

Several dynamic performance views provide information to help you tune the value of the PGA_AGGREGATE_TARGET parameter:

- V$SYSSTAT and V$SESSTAT
- V$PGASTAT
- V$PROCESS
- V$SQL_WORKAREA_ACTIVE
- V$SQL_WORKAREA

<table>
<thead>
<tr>
<th>View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V$SYSSTAT and V$SESSTAT</td>
<td>Statistics in the V$SYSSTAT and V$SESSTAT views show the total number of work areas executed with optimal memory size, one-pass memory size, and multi-pass memory size. These statistics are cumulative since the instance or the session was started.</td>
</tr>
<tr>
<td>V$PGASTAT</td>
<td>The V$PGASTAT view gives instance-level statistics (in bytes) on the PGA usage and the automatic memory manager.</td>
</tr>
<tr>
<td>V$PROCESS</td>
<td>The V$PROCESS view has one row per Oracle process connected to the instance. The columns PGA_USED_MEM, PGA_ALLOC_MEM and PGA_MAX_MEM can be used to monitor the PGA memory usage of these processes.</td>
</tr>
<tr>
<td>V$SQL_WORKAREA_ACTIVE</td>
<td>Query V$SQL_WORKAREA_ACTIVE to display the work areas that are active (or executing) in the instance. Small active sorts (under 64KB) are excluded from the view. Use this view to precisely monitor the size of all active work areas.</td>
</tr>
<tr>
<td>V$SQL_WORKAREA</td>
<td>Oracle maintains cumulative work area statistics for each loaded cursor whose execution plan uses one or more work areas. Every time a work area is deallocated, the V$SQL_WORKAREA table is updated with execution statistics for that work area.</td>
</tr>
</tbody>
</table>

Copyright © Oracle Corporation, 2002. All rights reserved.
V$SYSSTAT and V$SESSTAT

Find the total number and percentage of times work areas were executed in the optimal, one-pass, and multipass modes.

```
SELECT name profile, cnt,
    decode(total, 0, 0, round(cnt*100/total)) percentage
FROM (SELECT name, value cnt,
    (sum(value) over ()) total
FROM V$SYSSTAT
WHERE name like 'workarea exec%');
```

<table>
<thead>
<tr>
<th>PROFILE</th>
<th>CNT</th>
<th>PERCENTAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>workarea executions - optimal</td>
<td>5395</td>
<td>95</td>
</tr>
<tr>
<td>workarea executions - onepass</td>
<td>284</td>
<td>5</td>
</tr>
<tr>
<td>workarea executions - multipass</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**V$SYSSTAT and V$SESSTAT**

The query shown gives the total number and the percentage of times work areas are executed in the modes optimal, one-pass, and multipass, since the instance was started.

The output tells the DBA whether the value of the PGA_AGGREGATE_TARGET parameter must be changed. This can be changed with the ALTER SYSTEM statement.

Because the number of multipass executions is not zero, consider increasing the value of this parameter. If the value percentage of one-pass executions is high compared to optimal, increase the value of the PGA_AGGREGATE_TARGET parameter.

**Instructor Note**

Do not spend a lot of time of explaining these two views. These views are used by DBAs to help tune the database. Application developers must be aware of the existence of these views, but typically it is not an application developer’s task to modify the settings.

One-pass and multipass definitions are discussed on page 7.
Summary

In this lesson, you should have learned that the four most important processing phases are:

- Parse
- Bind
- Execute
- Fetch

Processing Phases

There are four main stages in processing a SQL statement:

- In the Parse stage, the Oracle9i Server checks the syntax of the statement, determines the execution plan if the statement is not already in the shared SQL area, and verifies the information in the data dictionary.
- In the Bind stage, any unresolved bind variables are assigned a value.
- In the Execute stage, the Oracle9i Server executes the statement performing required reads and writes (except for queries).
- In the Fetch stage (for queries only), rows are retrieved using an array fetch mechanism.
Summary

To optimize the use of shared SQL areas:

- Write generic code
- Follow coding standards
- Use bind variables
- Set the CURSOR_SHARING parameter
- Use automatic PGA memory management

Shared SQL Areas

To avoid parsing every SQL statement issued to the Oracle9i Server, develop a coding standard to take advantage of shared SQL areas:

- Follow coding standards for spacing, uppercase and lowercase, comments, object references, and so on. The main advantage comes from applications that use the same SQL statements.
- Use generic code, such as packaged procedures, database triggers, and so on.
- Use the same names to refer to identical database objects.

Try to use bind variables as opposed to literal values in your SQL statements, unless maximum performance for a single SQL statement is needed. There is a tradeoff between using shared SQL (with bind variables) and maximum performance (using literal values, enabling the optimizer to come up with an optimal execution plan). Additionally, use the automatic PGA memory management.
Practice Overview

This practice covers the following topics:

- SQL statement processing
- Using the V$LIBRARYCACHE view to analyze library cache performance
- Using the V$SQLAREA view to see information about all shared cursors in the cache

Instructor Note

This practice session is optional. Use it when you want to have a break between the initial (theoretical) lessons. Usually, you should try to go through the first seven lessons as quickly as possible, so that the first workshop can be done. During the three days of this class, the theory-to-practice ratio changes considerably.
Practice 2 (optional)

1. Log on with the user ID of SH and password SH. Your instructor will provide the connection information.

   Determine the arraysize set in your SQL*Plus environment.

2. Issue the following SQL statement, then view the information generated by this statement in the V$SQLAREA view.

   ```sql
   SQL> select channel_id, channel_desc
   2     from channels;
   SQL> select sql_text, sorts
   2     from   v$sqlarea
   3     where  command_type = 3
   4     and    lower(sql_text) like '%channels%';
   ```

3. Modify the query on the CHANNELS table and add a sort criterion. Then view the information generated by this statement in the V$SQLAREA view.

   ```sql
   SQL> select channel_id, channel_desc
   2     from   channels
   3     order  by channel_id;
   SQL> select sql_text, sorts
   2     from   v$sqlarea
   3     where  command_type = 3
   4     and    lower(sql_text) like '%channels%';
   ```

4. Insert a new record into the CHANNELS table. View the information generated by this statement in the V$SQLAREA view. Roll back your transaction to restore the original situation.

   ```sql
   SQL> insert into channels
   2     values('W', 'Wholesale', 'Direct');
   SQL> select sql_text, sorts
   2     from   v$sqlarea
   3     where  command_type in (2,3)
   4     and    lower(sql_text) like '%channels%';
   SQL> rollback;
   ```

5. Use the V$LIBRARYCACHE view to check the amount of SQL caching.
EXPLAIN and AUTOTRACE

Schedule: Timing  Topic
45 minutes    Lecture
10 minutes    Practice
55 minutes    Total

Instructor Note
This lesson introduces EXPLAIN and SQL*Plus AUTOTRACE. SQL*Plus AUTOTRACE is the diagnostic tool that is used in most workshops. Unfortunately, you must introduce statistics when using AUTOTRACE. Indicate that statistics will be discussed in later lessons and introduce the most important ones.

Keep the practice session short; there will be plenty of time to use the diagnostic tools in later lessons and workshops. The practice session is a short break between this lesson and the following one. You may also want to discuss the other diagnostic tools first, and combine the practice sessions.
Objectives

After completing this lesson, you should be able to do the following:

• Create a table named PLAN_TABLE
• Use EXPLAIN PLAN to show how a statement will be processed
• Query the performance view V$SQL_PLAN
• Use SQL*Plus AUTOTRACE to show SQL statement execution plans and statistics
Creating the Plan Table

- Run a standard script to create a plan table.

```
@utlxplan.sql
```

- Create your own plan table with a different name, by editing the `utlxplan.sql` script and changing the table name within the script.

Creating the Plan Table

The **EXPLAIN PLAN** command uses a table to store information about the execution plan chosen by the optimizer. If you examine the plan, you can see how the Oracle9i Server executes the statement. You must follow three steps to use **EXPLAIN PLAN**:

1. Create the plan output table.
2. Use the **EXPLAIN PLAN** command.
3. Retrieve the plan steps by using SQL statements against the plan table.

**The PLAN_TABLE**

You have several options in creating a plan table:

- You can use the default output table named `PLAN_TABLE`.
- You can use the `utlxplan.sql` script to create `PLAN_TABLE`.
  In a Windows environment, this script can be found in
  `%ORACLE_HOME%/rdbms/admin`; and under UNIX in
  `$ORACLE_HOME/rdbms/admin`. In a client-server environment, be sure to use the correct (server side) version of this script.
- You can also copy and edit the `utlxplan.sql` script; note that you should not change the structure of the table.

Oracle9i: SQL Tuning Workshop 3-3
The Create Plan Table Syntax

The entire syntax for the plan table is shown below:

```sql
create table PLAN_TABLE
  (statement_id       varchar2(30)
   ,timestamp          date
   ,remarks            varchar2(80)
   ,operation          varchar2(30)
   ,options            varchar2(255)
   ,object_node        varchar2(128)
   ,object_owner       varchar2(30)
   ,object_name        varchar2(30)
   ,object_instance    numeric
   ,object_type        varchar2(30)
   ,optimizer          varchar2(255)
   ,search_columns     number
   ,id                 numeric
   ,parent_id          numeric
   ,position           numeric
   ,cost               numeric
   ,cardinality        numeric
   ,bytes              numeric
   ,other_tag          varchar2(255)
   ,partition_start    varchar2(255)
   ,partition_stop     varchar2(255)
   ,partition_id       numeric
   ,other              long
   ,distribution       varchar2(30)
   ,cpu_cost           numeric
   ,io_cost            numeric
   ,temp_space         numeric);
```

Column descriptions can be found in Appendix B, “Diagnostic Tools Reference.” The most important columns will be discussed later in this lesson.

Instructor Note

The plan table creation syntax is for student interest only. Usually, the table is created using `utlxplan.sql`. However, if a participant wants to call the table by a different name, the full table definition is included here.
The **EXPLAIN PLAN** Command

This command inserts a row into the plan table for each step of the execution plan. In the syntax diagram shown in the slide, the fields in italics have the following meanings:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>text</em></td>
<td>This is an optional identifier for the statement. You should enter a value to identify each statement so that you can later specify the statement that you want explained. This is especially important when you share the plan table with others, or when you keep multiple execution plans in the same plan table.</td>
</tr>
<tr>
<td><em>schema.table</em></td>
<td>This is the optional name of the output table. The default is <code>PLAN_TABLE</code>.</td>
</tr>
<tr>
<td><em>statement</em></td>
<td>This is the text of the SQL statement.</td>
</tr>
</tbody>
</table>
EXPLAIN PLAN Example

This command inserts the execution plan of the SQL statement into the plan table, and adds the name tag demo01 for future reference.

Instructor Note

Demo script: demo03_01.sql

Note that this script could take some time to complete, because it must delete statistics (to force RBO) and to (re)create two indexes, to make the result consistent with the next slides.

Instructor Note for the next page

Demo script: demo03_02.sql (output on next slide)

Do not forget to execute “delete from plan_table” afterwards; you can also show the rp.sql script that automatically deletes all rows from the plan table.
## Displaying the Execution Plan

The select statement in the slide is designed to display the steps that would be taken if the SQL statement were executed. The following plan table columns are used:

- **STATEMENT_ID**: The value of the optional `STATEMENT_ID` parameter specified in the `EXPLAIN PLAN` statement.
- **OPERATION**: The name of the operation performed at this step; contains a value in the first row generated for each statement.
- **OPTIONS**: The options for the operation (For example, this will say “remote” if a remote table is referenced in the SQL statement).
- **OBJECT_NAME**: The name of the object.
- **ID**: The number of this step in the execution plan.
- **PARENT_ID**: The ID of the step that operates on the results of this step.
- **POSITION**: The order in which steps with the same parent are processed.

**Note**: Usually, you put this statement into a SQL script with a variable for the statement ID value. This makes using the explain plan much more feasible. In that same script, you can also delete (or truncate) all rows from the plan table after you have retrieved them.

Alternatively, use SQL*Plus `AUTOTRACE` to eliminate the need to issue queries against the plan table.

```sql
SQL> column "Query Plan" format a60

SQL> select id
2   , lpad(‘ ’, 2*level)||operation
3   ||decode(id,0,’ Cost = ’||position)
4   ||’ ’||options
5   ||’ ’||object_name as "Query Plan"
6   from plan_table
7   where statement_id = ‘demo01’
8   connect by prior id  = parent_id
9   start with id = 0;
```
Displaying the Execution Plan (continued)

Some other useful plan table columns (not used in the example) are:

- **TIMESTAMP**: The date and time when the `EXPLAIN PLAN` statement was issued.
- **COST**: The cost-based optimizer’s estimate of the cost of performing the operation, including the cost of any operations that are required to complete this operation (The value of this column does not have a particular unit of measurement. It is merely a weighted value used to compare costs of execution plans.)
- **CARDINALITY**: The cost-based optimizer’s estimate of the number of rows processed.
- **BYTES**: The cost-based optimizer’s estimate of the number of bytes.
- **OTHER**: The SQL text for remote cursors and parallel query slaves, as well as parallel query dataflow-operation information.
- **OTHER_TAG**: The function descriptions of the SQL text in the `OTHER` column.
- **OPTIMIZER**: The current mode of the optimizer.
- **OBJECT_OWNER**: The name of the schema that contains the table or index.
- **OBJECT_TYPE**: A modifier that describes the object, such as NON-UNIQUE for an index.

**Instructor Note**

A NULL value for **COST** always implies rule-based optimization.
Interpreting the Execution Plan

Each step of the execution plan either retrieves rows from the database or accepts rows as input from one or more other steps, also known as row sources. In the diagram, the numbers correspond to the ID values in the plan table (see previous page).

The AND_EQUAL operation (2) accepts the ROWIDs obtained by the scans of the STAT_INDEX (3) and LOC_INDEX (4), and returns the ROWIDs that come from both sources. This results in a set of ROWIDs that allow TABLE ACCESS (1) of the CLASSES table to return the rows that satisfy the query. ROWIDs are explained in the “Indexes and Basic Access Methods” lesson.

The cost of an execution plan is a value proportional to the expected elapsed time needed to execute the statement, using that execution plan. If the rule-based optimizer is used, then the cost is not available and shows a NULL value.
**Comparing Execution Plans**

You can determine the execution plan used by the optimizer by using `EXPLAIN PLAN`. If a SQL statement does not perform satisfactorily, you may decide to introduce an index. You should check the execution plan again to make sure that the index is used. You can also check the effect of using various optimizer hints, which are covered in the “Influencing the Optimizer” lesson.

**Reference**

For more information about interpreting execution plans, see *Oracle9i Database Performance Guide and Reference*, “Using `EXPLAIN PLAN`” and Appendix B of this student guide.
Using V$SQL_PLAN

- Provides a way of examining the execution plan for cursors that were recently executed.
- Information in this view is very similar to the output of an EXPLAIN PLAN statement:
  - EXPLAIN PLAN shows a theoretical plan that can be used if this statement were to be executed.
  - V$SQL_PLAN contains the actual plan used.

Using V$SQL_PLAN

This view provides a way of examining the execution plan for cursors that were recently executed.

The information in this view is very similar to the output of an EXPLAIN PLAN statement. However, EXPLAIN PLAN shows a theoretical plan that can be used if this statement were to be executed, whereas V$SQL_PLAN contains the actual plan used. The execution plan obtained by the EXPLAIN PLAN statement can be different from the execution plan used to execute the cursor, because the cursor might have been compiled with different values of session parameters (for example, HASH_AREA_SIZE).

V$SQL_PLAN shows the plan for a cursor, not for a SQL statement. The difference is that a SQL statement can have more than one cursor associated with it, with each cursor further identified by a CHILD_NUMBER. For example the same statement executed by different users will have different cursors associated with it if the object being referenced is in a different schema. Similarly different hints can cause different cursors. The V$SQL_PLAN table can be used to see the different plans for different child cursors of the same statement.
**V$SQL_PLAN Columns**

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>Address of the handle to the parent of the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECT#</td>
<td>Object number of the table or the index</td>
</tr>
<tr>
<td>OBJECT_NAME</td>
<td>Name of the table or index</td>
</tr>
<tr>
<td>OPERATION</td>
<td>Name of the internal operation performed in this step; for example, TABLE_ACCESS</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>A variation on the operation described in the OPERATION column; for example, FULL</td>
</tr>
<tr>
<td>OPTIMIZER</td>
<td>Current mode of the optimizer for the first row in the plan (statement line)</td>
</tr>
<tr>
<td>COST</td>
<td>Cost of the operation as estimated by the optimizer’s cost-based approach</td>
</tr>
</tbody>
</table>

**Using V$SQL_PLAN (continued)**

There are several uses for V$SQL_PLAN:

- Determining the current execution plan
- Identifying the effect of creating an index on a table
- Finding cursors containing a certain access path (for example, full table scan or index range scan)
- Identifying indexes that are, or are not, selected by the optimizer
- Determining whether the optimizer selects the particular execution plan (for example, nested loops join) expected by the developer

This view can also be used as a key mechanism in plan comparison to observe the changes in:

- Dropping or creating indexes
- Running the ANALYZE statement on the database objects
- Modifying initialization parameter values
- Switching from the rule-based optimizer to the cost-based optimizer
- Migrating the application or the database to a new release

If previous plans are kept, then it is then possible to identify how changes in the performance of a SQL statement can be correlated with changes in the execution plan for that statement.
### V$SQL_PLAN Columns

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH_VALUE</td>
<td>Hash value of the parent statement in the library cache</td>
</tr>
<tr>
<td>ADDRESS</td>
<td>Object number of the table or the index</td>
</tr>
<tr>
<td>CHILD_NUMBER</td>
<td>Child cursor number using this execution plan</td>
</tr>
<tr>
<td>POSITION</td>
<td>Order of processing for operations that all have the same PARENT_ID</td>
</tr>
<tr>
<td>PARENT_ID</td>
<td>ID of the next execution step that operates on the output of the current step</td>
</tr>
<tr>
<td>ID</td>
<td>A number assigned to each step in the execution plan</td>
</tr>
</tbody>
</table>

**Note:** This is a partial listing of columns.

The view contains almost all PLAN_TABLE columns, in addition to new columns. The columns that are also present in the PLAN_TABLE have the same values:

- ADDRESS
- HASH_VALUE

The two columns ADDRESS and HASH_VALUE can be used to join with V$SQLAREA to add the cursor-specific information.

The ADDRESS, HASH_VALUE and CHILD_NUMBER columns can be used to join with V$SQL to add the child cursor-specific information.
### Querying V$SQL_PLAN

```sql
SELECT id, lpad (' ', depth) || operation operation, options, object_name, optimizer, cost
FROM V$SQL_PLAN
WHERE hash_value = 912244748
AND address = '67D419DC'
START WITH id = 0
CONNECT BY
  ( prior id           = parent_id
    AND prior hash_value = hash_value
    AND prior child_number = child_number )
ORDER SIBLINGS BY id, position;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>OPERATION</th>
<th>OPTIONS</th>
<th>OBJECT_NAME</th>
<th>OPT</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td></td>
<td>CHO</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS</td>
<td>BY INDEX ROWID</td>
<td>PRODUCTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>AND-EQUAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>INDEX</td>
<td>RANGE SCAN</td>
<td>PRODUCTS_PROD_CAT_IX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>INDEX</td>
<td>RANGE SCAN</td>
<td>PRODUCTS_PROD_SUBCAT_I</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Querying V$SQL_PLAN

The above statement shows the execution plan for the SQL SELECT statement shown on page 3-6. Looking at the plan for a SQL statement is one of the first steps in tuning a SQL statement. The SQL statement that is used to return the plan is identified by the statement’s HASH_VALUE and ADDRESS from V$SQL_PLAN.

**Note:** For statements that use the rule-based approach, the COST column is null.

### Instructor Note

These are a few examples of how a SQL statement can result in more than one cursor:

When the same table name resolves to two separate tables:

- User1: `SELECT * FROM CUSTOMERS;
- User2: `SELECT * FROM CUSTOMERS;

Where User2 has his or her own customer table, and User1 uses the table referenced by a public synonym.
**SQL*Plus AUTOTRACE**

In SQL*Plus, you can automatically obtain the execution plan and some additional statistics on the running of a SQL command by using the AUTOTRACE setting. Unlike the EXPLAIN PLAN command, the statement is actually run, even if you choose to suppress the statement output; thus you can display realistic statistics. AUTOTRACE, a feature available since Oracle Server release 7.3, is an excellent diagnostic tool for SQL Statement Tuning. Because it is purely declarative, it is easier to use than EXPLAIN PLAN.

**Command Options**

- **OFF**  
  Disables autotracing SQL statements

- **ON**  
  Enables autotracing SQL statements

- **TRACEONLY**  
  Enables autotracing SQL statements and suppresses statement output

- **EXPLAIN**  
  Displays execution plans but does not display statistics

- **STATISTICS**  
  Displays statistics but does not display execution plans

**Note:** If both command options EXPLAIN and STATISTICS are omitted, execution plans and statistics are displayed by default.

**Instructor Note (for page 3-16)**

There are some more settings for the (optional) second part of AUTOTRACE output, for parallel and distributed queries. However, they are not relevant at this point.
Prerequisites for Using AUTOTRACE

To use the EXPLAIN option of AUTOTRACE, you must first create the PLAN_TABLE table in your schema, just as you would need to do before using the EXPLAIN PLAN command. To access STATISTICS data, you must have access to several dynamic performance tables. The DBA can grant access by using the role plustrace created in the plustrce.sql script. (This script name may vary by operating system.) The DBA must run the script as the SYS user and grant the role to users who want to use the STATISTICS option of AUTOTRACE.

Controlling the AUTOTRACE Execution Plan Layout

The execution plan consists of four columns displayed in the following order:

- **ID_PLUS_EXP**: The line number of each step
- **PARENT_ID_PLUS_EXP**: Parent step line number
- **PLAN_PLUS_EXP**: The report step
- **OBJECT_NODE_PLUS_EXP**: Database links or parallel query servers used

You can change the format of these columns, or suppress them, by using the SQL*Plus COLUMN command. For further information, see the Oracle9i SQL*Plus User’s Guide and Reference.
**SQL*Plus AUTOTRACE Statistics**

```sql
set autotrace traceonly statistics
select * from dual;
```

**Statistics**

<table>
<thead>
<tr>
<th>Count</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>recursive calls</td>
</tr>
<tr>
<td>2</td>
<td>db block gets</td>
</tr>
<tr>
<td>1</td>
<td>consistent gets</td>
</tr>
<tr>
<td>0</td>
<td>physical reads</td>
</tr>
<tr>
<td>0</td>
<td>redo size</td>
</tr>
<tr>
<td>367</td>
<td>bytes sent via SQL*Net to client</td>
</tr>
<tr>
<td>430</td>
<td>bytes received via SQL*Net from client</td>
</tr>
<tr>
<td>2</td>
<td>SQL*Net roundtrips to/from client</td>
</tr>
<tr>
<td>0</td>
<td>sorts (memory)</td>
</tr>
<tr>
<td>0</td>
<td>sorts (disk)</td>
</tr>
<tr>
<td>1</td>
<td>rows processed</td>
</tr>
</tbody>
</table>

**AUTOTRACE Statistics**

AUTOTRACE displays several statistics, not all of them relevant to the discussion at this stage. In the next lesson, when discussing SQL Trace and TKPROF, this course goes into more detail about SQL statement execution statistics. The most important ones are the following:

- **db block gets**: The number of logical I/Os for current gets
- **consistent gets**: The number of logical I/Os for read-consistent gets
- **physical reads**: The number of blocks read from disk
- **redo size**: The amount of redo generated (for DML statements)
- **sorts (memory)**: The number of sorts performed in memory
- **sorts (disk)**: The number of sorts performed using temporary disk storage

**Note:** Current gets are reads of the current block’s buffer cache; consistent gets are reads of rollback segments due to changes in the buffer. The db block gets, consistent gets, and physical reads are the three statistics that are usually monitored. These should be low compared to the number of rows retrieved. Sorts should be performed in memory rather than disk.

**Instructor Note**

Demo script: demo03_03.sql

See Instructor Note on page 3-19.
Summary

In this lesson, you should have learned how to:

- Create a `PLAN_TABLE`
- Use `EXPLAIN PLAN` to view execution plans
- Query `V$SQL_PLAN` to see the execution plan for cursors that were recently executed
- Use SQL*Plus `AUTOTRACE` to run statements and display execution plans and statistics

Summary

Tuning involves careful analysis of execution plans and execution statistics. Several tools are available to help you test and diagnose performance problems with a SQL statement:

- The `PLAN_TABLE` stores execution plans and can be created using the `utlxplan.sql` script.
- `EXPLAIN PLAN` determines how the optimizer processes a statement (the statement is not executed) and stores the execution plan in the `PLAN_TABLE`.
- You can also use the view `V$SQL_PLAN` to view the execution plan for recently executed cursors.
- The `SET AUTOTRACE` command in SQL*Plus enables you to view the execution plan and a variety of statistics about the execution whenever you execute a SQL statement.

You can use several other tools to monitor performance, alerting you to the need to tune SQL statements. Some of these tools will be covered in future lessons.
Practice Overview

This practice covers the following topics:

- Creating a PLAN_TABLE
- Using the EXPLAIN PLAN command to view execution plans
- Formatting execution plan results by using the rp.sql script
- Setting the SQL*Plus AUTOTRACE command to display execution plans and statistics for SQL statements

Instructor Note (for page 3-17)

Demo script: demo03_03.sql

Note that consistent gets can be higher than expected due to array processing; each new array needs one additional get. The number of array fetches equals (SQL*Net roundtrips - 2). AUTOTRACE performs an implicit query on v$sesstat; see script demo03_04.sql for a similar query on v$sesstat. This query shows the results for the session. demo03_03.sql shows the results from the last statement.

Instructor Note (for the Practice)

The optimizer_mode parameter for the session should be set to rule to ensure that the results are easier to interpret. At this point, the goal is to have the students use autotrace and understand basic execution plans.
Practice 3

1. Verify the existence of a PLAN_TABLE table in your schema. Create this table when needed, using the utlxplan.sql script. If the table already exists, ensure that the table is empty.

2. Ensure that your session is set to rule-based optimization by executing the following command. Then explain the following SQL SELECT statement:

   Note: Rule-based optimization is discussed in detail in a later lesson.

   SQL> alter session set optimizer_mode = RULE;
   SQL> select cust_first_name, cust_last_name
            2   from   customers
            3   where  cust_id = 100;

3. Now query the PLAN_TABLE table:

   SQL> select * from plan_table;

   Use the rp.sql script to query the PLAN_TABLE table in a more sophisticated manner:

   SQL> @rp

4. Enable SQL*Plus AUTOTRACE to display execution plans and run the command from step 2 again.

5. Set AUTOTRACE to suppress the command output and show both execution plans and statistics, and run step 2 again.

6. If you have some time left, enter your own SQL statements and experiment with EXPLAIN PLAN and AUTOTRACE.
This lesson introduces the SQL Trace utility and the associated initialization parameters. It explains how to enable SQL Trace for a session and how to find your trace files. It also covers the TKPROF command to format trace files.

Keep the practice session short; the students will have enough time to use SQL Trace and TKPROF in the workshops.
Objectives

After completing this lesson, you should be able to do the following:

- Configure SQL Trace to collect session statistics
- Enable SQL Trace and locate your trace files
- Format trace files using `TKPROF`
- Interpret the output of the `TKPROF` command

Background Reading

For more information about SQL Trace and `TKPROF`, see the following:

- Oracle9i Performance Methods
- Oracle9i Reference, “Initialization Parameters”

Oracle9i: SQL Tuning Workshop 4-2
A good way to compare two execution plans is to run them and compare the statistics to see which one performs better. You can use the SQL Trace facility to obtain performance information. SQL Trace writes its output to a file, and you use TKPROF to format it.

The SQL Trace facility:
- Can be switched on for an instance or a session
- Reports on volume and time statistics for the Parse, Execute, and Fetch phases
- Produces output that can be formatted by TKPROF

When the SQL trace facility is enabled for a session, Oracle generates a trace file containing statistics for traced SQL statements for that session. When the SQL trace facility is enabled for an instance, Oracle creates a separate trace file for each process.

**Note:** SQL Trace involves some overhead, so you usually do not want to enable SQL Trace at the instance level.
How to Use the SQL Trace Facility

1. Set the initialization parameters.
2. Switch on tracing.
3. Run the application.
4. Switch tracing off and format the trace file.
5. Interpret the output.

Running SQL Trace increases system overhead. Use SQL Trace only when required, and use it at the session level rather than at the instance level.

Instructor Note

On certain platforms you must exit your session after step 3 to get the full trace file on disk. Sometimes the last part is missing from a trace file; exiting your session resolves that problem. Statistics lines in the trace file are written only when the corresponding cursor is closed.

Note: Unlike previous releases TKPROF no longer needs to log back into the database to get an explain plan, because this is already provided in the trace files.
Initialization Parameters

Several initialization parameters relate to SQL Trace.

**TIMED_STATISTICS**

The SQL Trace facility provides a variety of information about processes, optionally including timing information. If you want timing information, you must turn on this parameter. You can do so for the entire database by setting the following initialization parameter in the parameter file before starting up and opening the database:

```
TIMED_STATISTICS = TRUE
```

The parameter also can be set dynamically for a particular session with the following command:

```
SQL> ALTER SESSION SET timed_statistics=true;
```

The timing statistics have a resolution of one-hundredth of a second. This means that any operation that finishes quickly may not be timed accurately, such as simple queries that execute quickly.

Having **TIMED_STATISTICS** turned on affects performance slightly because the Oracle server must do some additional work. Therefore, this parameter is commonly turned off until specifically desired for tuning purposes. However, keeping **TIMED_STATISTICS** turned on could make trace files more useful for support engineers during a system crash.

**Note:** The underlined values in the slide indicate the default values for the parameters.

---

**Initialization Parameters**

Several initialization parameters relate to SQL Trace.

**TIMED_STATISTICS**

The SQL Trace facility provides a variety of information about processes, optionally including timing information. If you want timing information, you must turn on this parameter. You can do so for the entire database by setting the following initialization parameter in the parameter file before starting up and opening the database:

```
TIMED_STATISTICS = {false|true}
```

The parameter also can be set dynamically for a particular session with the following command:

```
SQL> ALTER SESSION SET timed_statistics=true;
```

The timing statistics have a resolution of one-hundredth of a second. This means that any operation that finishes quickly may not be timed accurately, such as simple queries that execute quickly.

Having **TIMED_STATISTICS** turned on affects performance slightly because the Oracle server must do some additional work. Therefore, this parameter is commonly turned off until specifically desired for tuning purposes. However, keeping **TIMED_STATISTICS** turned on could make trace files more useful for support engineers during a system crash.

**Note:** The underlined values in the slide indicate the default values for the parameters.
Initialization Parameters (Continued)

**MAX_DUMP_FILE_SIZE** and **USER_DUMP_DEST**

These two parameters control the size and destination of the output file:

\[
\begin{align*}
\text{MAX\_DUMP\_FILE\_SIZE} &= n \\
\text{USER\_DUMP\_DEST} &= \text{directory\_path}
\end{align*}
\]

The **MAX\_DUMP\_FILE\_SIZE** default value is 500, and this parameter value is expressed in operating system blocks. **MAX\_DUMP\_FILE\_SIZE** can also be changed at the session level by using the **ALTER SESSION** command.

The default **USER\_DUMP\_DEST** location varies by operating system; it typically is the default destination for system dumps on your system. **USER\_DUMP\_DEST** is a system-level parameter that cannot be changed at the session level. It can only be changed dynamically by a database administrator using the **ALTER SYSTEM** command.

**Obtain Information about Parameter Settings**

You can display current parameter values by querying the **V\_PARAMETER** view:

```
SQL> SELECT name, value 
2  FROM v$parameter 
3  WHERE name LIKE '%dump%';
```

**Note:** The directory path specifications that show up in the **VALUE** column are operating system dependent. This view is accessible only by users with DBA privileges.

**Instructor Note**

You can also use the **show parameter <search string>** command; all commands that were specific for the Server Manager tool are now available in SQL*Plus, provided you have the correct privileges.

If the **MAX\_DUMP\_FILE\_SIZE** is reached, *** Trace File Full *** is written to the trace file. No notification or error message is displayed, and the background process stops writing to the trace file.

Note that there is an additional initialization parameter, **SQL\_TRACE**, that influences SQL Trace. However, it is not a parameter to use when preparing for SQL Trace, but rather when enabling SQL Trace. That is why it is not on the previous slide but is introduced on the next slide. You should discourage use of that parameter and suggest enabling SQL Trace at the session level instead.

Note that when you are using SQL Trace in Multi-Threaded Server (MTS), the shared server processes will create trace files in the **BACKGROUND\_DUMP\_DEST**, not in the **USER\_DUMP\_DEST** directory. It is recommended to use a dedicated server process for tuning.
Switching On SQL Trace

- For an instance, set the following parameter:

```
SQL_TRACE = TRUE
```

- For your current session:

```
SQL> ALTER SESSION SET sql_trace = true;
```

```
SQL> EXECUTE dbms_session.set_sql_trace(true);
```

- For any session:

```
SQL> EXECUTE dbms_system.set_sql_trace_in_session
    2  (session_id, serial_id, true);
```

Switching On SQL Trace for an Instance

In order to switch on SQL Trace for an entire instance, set the initialization parameter:

```
SQL_TRACE = TRUE
```

Switching On SQL Trace for a Session

You can use a SQL command to switch on SQL Trace for your session.

```
SQL> ALTER SESSION SET sql_trace = true;
```

You can also use a supplied package to switch on SQL Trace for your session. This is useful when you want to turn SQL Trace on and off from within a PL/SQL unit.

```
SQL> EXECUTE dbms_session.set_sql_trace(true);
```

You can also switch on SQL Trace for another user’s session with a supplied package.

```
SQL> EXECUTE dbms_system.set_sql_trace_in_session
    2  (session_id, serial_id, true);
```

In this procedure call, `session_id` and `serial_id` are the values in the SID and SERIAL# columns of `V$SESSION`, a data dictionary view commonly used by database administrators.

Switching Off SQL Trace

When you have finished tuning, you can switch off SQL Trace by using any of the above methods, substituting the word FALSE for TRUE. If you switch on SQL Trace for a single session, exiting that session also turns off SQL Trace.
Finding Your Trace Files

- Look in the directory specified by \texttt{USER\_DUMP\_DEST}.
- If only a few people are running SQL Trace, look for the most recent time stamp.
- Otherwise, consider writing a script.

\begin{verbatim}
SQL> @readtrace.sql

SQL> EXECUTE gettrace;
PL/SQL procedure successfully completed.
\end{verbatim}

Finding Your Trace Files

Identifying your trace file in the directory specified by \texttt{USER\_DUMP\_DEST} is usually a matter of finding the trace file with the most recent time stamp. However, this task becomes more difficult when several users generate trace files at the same time.

When Several Users Create Trace Files

When several users create trace files simultaneously, the \texttt{readtrace.sql} script can be useful. It creates a procedure that opens your current trace file by using the \texttt{UTL\_FILE} package. The default output file name is \texttt{username.trc}, but you can also specify a name yourself.

```
SQL> @readtrace.sql
SQL> ALTER SESSION SET sql_trace = true;
SQL> SELECT * FROM dual;
SQL> execute gettrace('output_filename');
```

Instructor Note

Tell the students that \texttt{readtrace} is \textit{not} a standard Oracle utility that comes with the software, but is only developed and provided for this course. The script is written for an NT environment and must be modified for other environments.

By default, the \texttt{.trc} file is named after the \texttt{SPID}. You can find the \texttt{SPID} in \texttt{v$process}. To find your \texttt{SPID} for the session, run the \texttt{demo04_01spid.sql} script.

\textbf{Oracle9i: SQL Tuning Workshop 4-8}
Formatting Your Trace Files

Use the TKPROF command to format your trace files into a readable output.

This is the TKPROF syntax:

```
OS> tkprof tracefile outputfile [options]
```

- `tracefile` is the name of the trace output file (input for TKPROF).
- `outputfile` is the name of the file to store the formatted results.

When the TKPROF command is entered without any arguments, it generates a usage message together with a description of all TKPROF options. See the next page for a full listing.

**Note:** Trace files generated immediately after instance startup contain data that reflects the activity of the startup process. In particular, they reflect a disproportionate amount of physical I/O activity as caches in the SGA are filled. For the purposes of tuning, you should ignore such trace files.

**Instructor Note**

These are just some examples; the full TKPROF syntax will be explained later. In previous releases, TKPROF had a version component in the command name on Windows platforms (for example, tkprof73 or tkprof80). With Oracle8i and higher, there is no tkprof8i command both in UNIX and NT anymore.

**Instructor Note (for page 4-10)**

This is the (formatted) output you get when executing the TKPROF command without arguments; some options are explained in more detail on the following pages.
Output of the **TKPROF** Command

This is the output you get when you enter the **TKPROF** command without any arguments:

**Usage:**

```
tkprof tracefile outputfile [explain= ] [table= ] [print= ] [insert= ] [sys= ] [sort= ]
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>table=schema.tablename</code></td>
<td>Use schema.tablename with the explain option.</td>
</tr>
<tr>
<td><code>explain=user/password</code></td>
<td>Connect to the database and issue EXPLAIN PLAN.</td>
</tr>
<tr>
<td><code>print=integer</code></td>
<td>List only the first integer SQL statements.</td>
</tr>
<tr>
<td><code>aggregate=no</code></td>
<td>Do not aggregate identical SQL statements into one result.</td>
</tr>
<tr>
<td><code>insert=filename</code></td>
<td>List SQL statements and data inside INSERT statements.</td>
</tr>
<tr>
<td><code>sys=no</code></td>
<td><strong>TKPROF</strong> does not list SQL statements run as user <strong>SYS</strong>.</td>
</tr>
<tr>
<td><code>record=filename</code></td>
<td>Record nonrecursive statements found in the trace file.</td>
</tr>
<tr>
<td><code>sort=option</code></td>
<td>A set of zero or more of the following sort options:</td>
</tr>
</tbody>
</table>

**Options:**

- `prsctt` Number of times parse was called
- `prscpu` CPU time parsing
- `prsla` Elapsed time parsing
- `prsdsk` Number of disk reads during parse
- `prsqry` Number of buffers for consistent read during parse
- `prscu` Number of buffers for current read during parse
- `prsmis` Number of misses in library cache during parse
- `execnt` Number of execute was called
- `execpu` CPU time spent executing
- `exeela` Elapsed time executing
- `exedsk` Number of disk reads during execute
- `exeqry` Number of buffers for consistent read during execute
- `execu` Number of buffers for current read during execute
- `exerow` Number of rows processed during execute
- `exemis` Number of library cache misses during execute
- `fchcnt` Number of times fetch was called
- `fchcpu` CPU time spent fetching
- `fchela` Elapsed time fetching
- `fchdsk` Number of disk reads during fetch
- `fchqry` Number of buffers for consistent read during fetch
- `fchcu` Number of buffers for current read during fetch
- `fchrow` Number of rows fetched
- `userid` ID of user who parsed the cursor
TKPROF Command Options

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SORT = option</td>
<td>The order in which to sort the statements in the report (See previous page for a list of values.)</td>
</tr>
<tr>
<td>PRINT = n</td>
<td>Produces a report on this (sorted) number of statements only (This option is especially useful in combination with the SORT option.)</td>
</tr>
<tr>
<td>EXPLAIN = username/password</td>
<td>Logs on and executes EXPLAIN PLAN in the specified schema</td>
</tr>
<tr>
<td>INSERT = filename</td>
<td>Creates a SQL script to load the TKPROF results into a database table</td>
</tr>
<tr>
<td>SYS = NO</td>
<td>Disables the listing of recursive SQL statements, issued by the user SYS</td>
</tr>
<tr>
<td>AGGREGATE = NO</td>
<td>Disables or enables the (default) behavior of TKPROF, aggregating identical SQL statements into one record</td>
</tr>
<tr>
<td>RECORD = filename</td>
<td>Creates a SQL script with all the nonrecursive SQL statements found in the trace file (This script can be used to replay the tuning session later.)</td>
</tr>
<tr>
<td>TABLE = schema.tablename</td>
<td>Specifies the table to temporarily store execution plans before writing them to the output file (This parameter is ignored if EXPLAIN is not specified. It is useful when several individuals use TKPROF to tune the same schema concurrently, to avoid destructive interference.)</td>
</tr>
</tbody>
</table>
Output of the **TKPROF** Command

- **Text of the SQL statement**
- **Trace statistics** (for statement and recursive calls) separated into three SQL processing steps:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARSE</td>
<td>Translates the SQL statement into an execution plan</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>Executes the statement (This step modifies the data for <strong>INSERT</strong>, <strong>UPDATE</strong>, and <strong>DELETE</strong> statements.)</td>
</tr>
<tr>
<td>FETCH</td>
<td>Retrieves the rows returned by a query (Fetches are performed only for <strong>SELECT</strong> statements.)</td>
</tr>
</tbody>
</table>

**Note:** The **PARSE** value includes both *hard* and *soft* parses. A hard parse refers to the development of the execution plan (including optimization); it is subsequently stored in the library cache. A soft parse means that a SQL statement is sent for parsing to the kernel, but the kernel finds it in the library cache, and only needs to verify things such as access rights. Hard parses can be expensive, in particular due to the optimization; a soft parse is mostly expensive in terms of library cache activity.
Output of the **TKPROF** Command

Seven categories of trace statistics:

<table>
<thead>
<tr>
<th>Count</th>
<th>Number of times procedure was executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Number of seconds to process</td>
</tr>
<tr>
<td>Elapsed</td>
<td>Total number of seconds to execute</td>
</tr>
<tr>
<td>Disk</td>
<td>Number of physical blocks read</td>
</tr>
<tr>
<td>Query</td>
<td>Number of logical buffers read for consistent read</td>
</tr>
<tr>
<td>Current</td>
<td>Number of logical buffers read in current mode</td>
</tr>
<tr>
<td>Rows</td>
<td>Number of rows processed by the fetch or execute</td>
</tr>
</tbody>
</table>
**Output of the TKPROF Command (continued)**

Next to the CALL column, TKPROF displays the following statistics for each statement:

<table>
<thead>
<tr>
<th>Count</th>
<th>Number of times a statement was parsed, executed, or fetched (Check this column for values &gt;1 before interpreting the statistics in the other columns. Unless the AGGREGATE = NO option is used, TKPROF aggregates identical statement executions into one summary table.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Total CPU time in seconds for all parse, execute, or fetch calls</td>
</tr>
<tr>
<td>Elapsed</td>
<td>Total elapsed time in seconds for all parse, execute, or fetch calls</td>
</tr>
<tr>
<td>Disk</td>
<td>Total number of data blocks physically read from the data files on disk for all parse, execute, or fetch calls</td>
</tr>
<tr>
<td>Query</td>
<td>Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls (Buffers are usually retrieved in consistent mode for queries.)</td>
</tr>
<tr>
<td>Current</td>
<td>Total number of buffers retrieved in current mode (Buffers typically are retrieved in current mode for DML statements. However, segment header blocks are always retrieved in current mode.)</td>
</tr>
<tr>
<td>Rows</td>
<td>Total number of rows processed by the SQL statement (This total does not include rows processed by subqueries of the SQL statement. For SELECT statements, the number of rows returned appears for the fetch step. For UPDATE, DELETE, and INSERT statements, the number of rows processed appears for the execute step.)</td>
</tr>
</tbody>
</table>

**Note:**

- DISK is equivalent to physical reads from v$sysstat or AUTOTRACE.
- QUERY is equivalent to consistent gets from v$sysstat or AUTOTRACE.
- CURRENT is equivalent to db block gets from v$sysstat or AUTOTRACE.
Output of the TKPROF Command

The TKPROF output also includes the following:

- Recursive SQL statements
- Library cache misses
- Parsing user ID
- Execution plan
- Optimizer mode or hint

Recursive Calls
Sometimes in order to execute a SQL statement issued by a user, the Oracle server must issue additional statements. Such statements are called recursive SQL statements. For example, if you insert a row into a table that does not have enough space to hold that row, the Oracle server makes recursive calls to allocate the space dynamically. Recursive calls are also generated when data dictionary information is not available in the data dictionary cache and must be retrieved from disk.

If recursive calls occur while the SQL trace facility is enabled, TKPROF marks them clearly as recursive SQL statements in the output file. You can suppress the listing of recursive calls in the output file by setting the SYS=NO command-line parameter. Note that the statistics for recursive SQL statements are always included in the listing for the SQL statement that caused the recursive call.

Library Cache Misses
TKPROF also lists the number of library cache misses resulting from parse and execute steps for each SQL statement. These statistics appear on separate lines following the tabular statistics.

Parsing User ID
This is the ID of the last user to parse the statement.
Execution Plan
If you specify the EXPLAIN parameter on the TKPROF command line, TKPROF uses the EXPLAIN PLAN command to generate the execution plan of each SQL statement traced. TKPROF also displays the number of rows processed by each step of the execution plan.

Note: Be aware that the execution plan is generated at the time that the TKPROF command is run and not at the time the trace file was produced. This could make a difference if, for example, an index has been created or dropped since tracing the statements.

Optimizer Mode or Hint
This indicates the optimizer hint that is used during the execution of the statement. If there is no hint, then it shows the optimizer mode that is used. Hints and optimizer modes are discussed in more detail in the “Influencing the Optimizer” lesson.

Instructor Note
Note that TKPROF output also shows the row source operators (RSO) before the (optional) full execution plan. Note that these row source operators are derived from the trace file STAT lines without accessing the database, as opposed to the additional execution plan generated with the EXPLAIN option.
Example of **TKPROF** Output: No Index

The example in the slide shows 234 rows being fetched from the `CUSTOMERS` table. It requires 4835 reads and 2603.73 seconds of CPU fetch time. The statement is executed through a full table scan of the `CUSTOMERS` table, as you can see using the `EXPLAIN` command-line option.

The statement must be optimized.

**Note:** If `cpu` or `elapsed` values are 0, then `timed_statistics` is not turned on.

**Instructor Note**

Demo script: `demo04_02.sql`

Note that this script could take some time to complete (approximately 3 minutes).

You are prompted from the script `dai.sql` to drop all indexes. Enter `CUSTOMERS` when prompted.

After running the script, go to the OS, and use the `TKPROF` command to format the results.
**Example of TKPROF Output: Unique Index**

The same row is retrieved, but in this case I/O is substantially reduced to only 87 blocks read. These results are achieved because the statement uses an index.

You can achieve significant improvements in performance by sensible indexing. Identify areas for potential improvement using the SQL Trace facility.

**Note:** Indexes should not be built until required. Indexes slow down the processing of INSERT, UPDATE, and DELETE commands, because references to rows must be added, changed, or removed. Unused indexes should be removed. If possible, process all application SQL through EXPLAIN PLAN and remove any indexes that are not referenced.

**Instructor Note**

This demonstration can be done using either SQL*Plus AUTO TRACE or TKPROF:

Demo script: demo04_03.sql

Note that this script may take some time to complete (approximately 3 minutes).

This script creates an index on the CUST_LAST_NAME column.

After running the script, go to the OS, and use the TKPROF command to format the results.
Some **TKPROF** Interpretation Pitfalls

- Read consistency trap
- Schema trap
- Time trap
- Trigger trap

---

**Read Consistency Trap**

Sometimes other transactions hold uncommitted changes against a table. This increases the number of block visits because additional blocks must be constructed and visited for read consistency.

**Schema Trap**

The **TKPROF** statistics show a high number of block visits whereas the execution plan indicates an index access. Especially if the output shows a nonzero value for current, the table is probably accessed by a full table scan (compare the current columns on the two previous pages). The index is probably built after the trace file is produced, or the table and column statistics may be recomputed.

**Time Trap**

If a particular DML statement shows high timing statistics, the explanation may be interference from another transaction holding (shared) locks on the table. That is why CPU time is a better indicator than elapsed time.

**Trigger Trap**

The resources reported for a statement include those for all of the SQL issued while the statement was being processed. Therefore, they include any resources used within a trigger, along with the resources used by any other recursive SQL (such as SQL used for space allocation).
Summary

In this lesson, you should have learned how to:

• Set SQL Trace initialization parameters
  – SQL_TRACE, TIMED_STATISTICS
  – USER_DUMP_DEST, MAX_DUMP_FILE_SIZE

• Switch on SQL Trace for a session

  ALTER SESSION set sql_trace = true
  dbms_session.set_sql_trace(...)
  dbms_system.set_sql_trace_in_session(...)

• Format trace files with TKPROF

• Interpret the output
Practice Overview

This practice covers the following topics:

- Viewing and changing initialization parameters
- Using the SQL Trace facility
- Configuring SQL Trace
- Using TKPROF to format trace files
- Interpreting TKPROF results

Practice Overview

In this practice, you view and change initialization parameters that influence tracing SQL statements. Then you use the SQL_TRACE facility to analyze a SQL statement. The results of the analysis are written to a trace file. Find the trace file and use TKPROF to format the trace statistics. Interpret the formatted trace results.
Practice 4

1. Check the following initialization parameters:
   - TIMED_STATISTICS
   - MAX_DUMP_FILE_SIZE
   - USER_DUMP_DEST
   - SQL_TRACE

Make sure that both TIMED_STATISTICS and SQL_TRACE are set to TRUE for your session. Turn SQL*Plus AUTOTRACE off.

2. Execute the following SQL command:
   ```sql
   SQL> select prod_name, prod_desc, supplier_id
   2     from products
   3    where prod_id = 200;
   ```

3. Now disable tracing for your session and try to find your trace file. Open the trace file with an operating system text editor and investigate the contents of your trace file.

   **Note:** The edit command in SQL*Plus will default to the "ex" editor. To use vi, type the command `define_editor=vi` in your SQL*Plus session.

4. Start TKPROF without command-line arguments. TKPROF displays a usage message, listing all command-line options.

5. Start TKPROF again, this time with the name of your trace file as an argument. Also specify a name for the TKPROF output report. Check the result with an operating system editor again, and see how the readability is improved.

   **Note:** If you are using UNIX to edit the file, use the command `vi filename`.

6. Finally, start TKPROF in such a way that execution plans are added to the report and recursive SQL statements are suppressed.
Rule-Based Optimization  
Versus Cost-Based Optimization

Schedule: Timing       Topic
      30 minutes   Lecture
      15 minutes   Practice (optional)
      45 minutes   Total

Instructor Note
This short lesson introduces both the rule-based optimizer and the cost-based optimizer, and discusses how to influence the optimizer mode at the instance and the session level. Do not spend too much time on the rule-based optimizer.
Objectives

After completing this lesson, you should be able to do the following:

• Describe the functions of the Oracle9i optimizer
• Distinguish between rule-based optimization (RBO) and cost-based optimization (CBO)
• Identify when RBO and CBO are used
• Identify the factors that CBO considers
• Identify and influence RBO behavior
• Set the optimizer approach at the instance and session level

Instructor Note

This lesson introduces the Oracle9i optimizer and explains the differences between RBO and CBO and the default behavior when choosing between RBO and CBO. This is the only lesson that specifically addresses RBO behavior. Statistics collection and how to influence CBO are covered later in the course.
Overview

- Rule-based optimization (RBO)
- Cost-based optimization (CBO)
- Setting an optimizer approach
- The RBO ranking scheme
- Influencing the RBO

Overview

There are various methods and access paths that can be used to execute a SQL statement. It is the optimizer’s job to choose which ones to use.

- RBO decides based on a set of rules.
- CBO bases this choice on statistics that it holds about the tables.

This lesson explains how the Oracle optimizer decides between these two approaches.

To use cost-based optimization, you should collect statistics about the tables involved. This is covered in a later lesson. Cost-based optimization can be influenced in several sophisticated ways; this is also covered in a later lesson.

Although it is still possible to use rule-based optimization, you should not use this method. This lesson shows how rule-based optimization uses a ranking scheme to produce execution plans. You will also see coding techniques that have been popular in the past to influence rule-based optimization.

Finally, this lesson details the Oracle9i optimizer’s four basic approaches, which can be set at the instance and session level.
Functions of the Oracle9i Optimizer

- Evaluates expressions and conditions
- Transforms statements into equivalent statements
- Decides how to access the data
- Decides how to join tables
- Decides which path is most efficient

The optimizer is the part of the Oracle9i Server that creates the execution plan for a SQL statement. An execution plan is a series of operations that are performed in sequence to execute the statement. Most of these steps were discussed in the “SQL Statement processing” lesson.

The optimizer uses various pieces of information to work out the best path:

- Hints supplied by the developer
- Statistics
- Information in the dictionary
- The WHERE clause

Usually, the optimizer works in the background. However, with diagnostic tools such as EXPLAIN, and SQL*Plus AUTOTRACE, you can see the decisions that the optimizer makes.
Rule-Based Optimization

- Is supported for backward compatibility
- Is syntax driven
- Uses no statistics about the data
- Calculates no costs

Rule-based optimization is supported in Oracle9i, but only for compatibility reasons with Oracle version 6 and earlier. If you still have existing OLTP applications developed and carefully tuned using Oracle version 6, you might want to continue using rule-based optimization when you upgrade these applications to Oracle9i.

Rule-based optimization is syntax driven, which means that changing the syntax of SQL statements could change the performance. Be aware of the following:

- No statistics are used, so the optimizer does not know anything about the number of rows in a table and the values in the table columns.
- No costs of execution plans are calculated; the decision about an execution plan is fully based on a set of rules.

Note: When using diagnostic tools such as EXPLAIN and SQL*Plus AUTOTRACE, the rule-based optimization approach is easy to recognize: the COST will always display a NULL value. As soon as a COST value appears, you know that the cost-based optimizer approach is used.
Cost-Based Optimization

- Was introduced with Oracle7
- Is statistics driven
- Bases decisions on cost calculation
  - Number of logical reads
  - Network transmissions
- Is continuously enhanced
- Typically outperforms RBO
- Estimates CPU usage. In Oracle9i cost includes estimated usage of CPU for SQL functions and operators.

Cost-Based Optimization

Oracle7 was the first release that implemented cost-based optimization. Unlike rule-based optimization, the cost-based optimizer bases its decisions on cost estimates, which in turn are based on statistics stored in the data dictionary.

In calculating the cost of execution plans, cost-based optimization considers the following:

- The number of logical reads (This is the most important factor.)
- CPU utilization
- Network transmissions

Cost-based optimization is continually enhanced with each new Oracle server release, and several newer features—such as hash joins, star queries, histograms, and index-organized tables—are available only to cost-based optimization.

With Oracle9i the estimated usage of CPU for SQL functions and operators is included in the overall estimate of computer resources usage, together with disk I/O and memory, and is used to compute the cost of access paths and join orders. Estimate of network usage is also included when data is shipped between query servers running on different nodes.

The result is an improved accuracy of the cost and size models used by the query optimizer. This helps the optimizer produce better execution plans, improving query performance.
Cost-Based Query Optimization Challenges

The task of the optimizer is to bridge the gap between WHAT is specified in a SQL statement and HOW this can be solved by the execution engine of the database.

For a single SQL statement there can be a huge number of alternative execution plans that all provide the correct answer but differ significantly in execution time. The challenge for the optimizer is to quickly find a cost-effective plan. Here it should balance the search time and the predicted execution time to find the best plan so far.

To estimate the cost of an execution plan it uses a costmodel and the obtained statistics. The costmodel consists of a costfunction for each possible operator of an execution plan and a mechanism for estimating the intermediate result sizes.

The problem of query optimization is known to be a mathematically complex problem. And because the decisions made by the optimizer are based on statistics and estimations it can never guarantee to find the most cost-effective plan possible. At best it can try to find a reasonably cost-effective plan within a reasonable amount of time.
Choosing Between RBO and CBO

- **Oracle9i default optimizer behavior:**
  - RBO when no statistics are available
  - CBO when statistics are available for at least one referenced object
- **Behavior can be influenced at the following levels:**
  - Instance level
  - Session level
  - Statement level

Influencing the Default Behavior

This default behavior can be influenced—for example, to force rule-based optimization under all circumstances—at three different levels:

- Instance level
- Session level
- Statement level

Influencing the optimizer at the statement level is discussed in the “Influencing the Optimizer” lesson.

**Note:** If any table referenced in a query has a degree of parallelism greater than 1, CBO will always be used regardless of any hints or settings. Also make sure that you have table statistics, otherwise collecting column statistics for a table will not result in cost-based optimization.
Setting the Optimizer Approach

- At the instance level, set the following parameter:

\[
\text{OPTIMIZER\_MODE} = \{\text{CHOOSE}|\text{RULE}|\text{FIRST\_ROWS}\_n|\text{ALL\_ROWS}\}
\]

- For a session, use the following SQL command:

\[
\text{ALTER SESSION SET optimizer\_mode} = \{\text{choose}|\text{rule}|\text{first\_rows}(\_n)|\text{all\_rows}\}
\]

Oracle9i has four basic optimizer approaches:

- **CHOOSE**
  Behaves as described earlier in this lesson: cost-based when statistics are available, rule-based when they are not (which is the default value)

- **RULE**
  Forces rule-based optimization regardless of the presence of statistics

- **FIRST\_ROWS\_[\_n]**
  Optimizes to reduce time until the first result comes back (such as for interactive applications) \(n=1, 10, 100, \text{or 1000}\)

- **ALL\_ROWS**
  Optimizes overall throughput (such as for reports and batch jobs)

**Optimizer Approach Levels**

At the instance level, you can set the optimizer approach by using the `OPTIMIZER\_MODE` initialization parameter. When this parameter is not set, it defaults to `CHOOSE`. When statistics are available, `CHOOSE` is equivalent to `ALL\_ROWS`.

At the session level, the optimizer approach set at the instance level can be overruled by using the `ALTER SESSION` command.

**Instructor Note**

For backward compatibility reasons, `OPTIMIZER\_GOAL` is still supported as a synonym for `OPTIMIZER\_MODE` in the `ALTER SESSION` command. However, `OPTIMIZER\_GOAL` is no longer documented in Oracle9i. There is no need to mention this to the students.
First Rows Optimization

OPTIMIZER_MODE =
  • FIRST_ROWS_1
  • FIRST_ROWS_10
  • FIRST_ROWS_100
  • FIRST_ROWS_1000

ALTER SESSION
  SET OPTIMIZER_MODE = FIRST_ROWS_n
/**+ FIRST_ROWS(x) */

New First Rows Optimization

Oracle9i introduces a new way of doing first rows optimization. The old mode was partially based on heuristics, such as always using an index if possible. These heuristics sometimes led to bad plans.

The new approach is completely cost based and allows optimizing for a particular number of first rows, for example, the first 10 rows.

It is invoked as an argument to the optimizer_mode initialization parameter or the optimizer_goal session parameter, which allows the following values:

  • FIRST_ROWS_1
  • FIRST_ROWS_10
  • FIRST_ROWS_100
  • FIRST_ROWS_1000

In addition, the FIRST_ROWS hint now takes a numeric argument that is not limited to the values for the parameter. For instance, you could specify /**+ FIRST_ROWS(20) */ as a hint.

Without a numeric argument, both the parameter and the hint imply the old first rows behavior, which is retained for backwards compatibility and plan stability.
Rule-Based Optimization

- Uses indexes unconditionally, because it has no awareness of statistics
- Uses a built-in ranking scheme with scores for access paths
- Is difficult to influence

Influencing RBO

You cannot influence rule-based optimization, except by creating indexes, dropping indexes, or making indexes unusable by changing the statement syntax.

If the RBO finds two execution plans that score the same on the ranking scheme, it makes an arbitrary decision. For example, suppose each available plan can use only one of two different indexes. In that case the RBO decision is not documented, but dropping and recreating one of the involved indexes will influence the RBO behavior: it will use the newly created index. Thus, you should be aware that recreating a series of indexes in a different order will probably influence the RBO behavior. Alternatively, you can change the SQL statement to make it impossible for the RBO to use one of the indexes. This is discussed later in this lesson.
RBO Ranking Scheme

Rule-based optimization applies rules based on the statement syntax to establish the access path. The optimizer determines all available access paths for a statement and then chooses the path with the lowest ranking number from the ranking scheme displayed in the slide.

This approach always assumes that a full table scan (rank 15) is the worst access method. This might not actually be the case, especially with small tables or with queries that return a large percentage of the rows in a table.

These (and more) access paths are also available to cost-based optimization. However, the optimizer ignores the rankings and determines the expected cost for each possible access path, then chooses the one with the smallest estimated cost. Many features (such as hash joins, star queries, histograms, and index-organized tables) are available only through cost-based optimization.

Note: For more information about RBO access paths, see the lesson titled “The Optimizer” in Oracle9i Tuning and Oracle9i Concepts.

Instructor Note (for page 5-13)

Demo script: demo05_01.sql.

This script creates indexes on LAST_NAME, COUNTRY_ID, and CREDIT_LIMIT, and forces RBO.
Rule-Based Optimization Example

The rule-based approach chooses a path based on the syntax of the statement. The optimizer chooses the access path with the lowest ranking number.

If the CUST_LAST_NAME, COUNTRY_ID, and CUST_CREDIT_LIMIT columns are indexed, then the rule-based optimizer has the following choices:

<table>
<thead>
<tr>
<th>Access Path</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full table scan</td>
<td>15</td>
</tr>
<tr>
<td>ORDER BY on indexed column</td>
<td>14</td>
</tr>
<tr>
<td>COUNTRY_ID index equality (range) scan</td>
<td>9</td>
</tr>
<tr>
<td>CUST_CREDIT_LIMIT index bounded-range search</td>
<td>10</td>
</tr>
</tbody>
</table>

The lowest rank is 9, so the index on COUNTRY_ID will be used to access the rows of the CUSTOMERS table. Note that although the index on COUNTRY_ID is used, a filter operation must be performed on the CUST_CREDIT_LIMIT column to return the correct results.
Influencing Rule-Based Optimization

- Change the table order in the \texttt{FROM} clause
- Change the predicate order in the \texttt{WHERE} clause
- Create or drop indexes
- Suppress index usage with syntax tricks:
  - \texttt{numeric\_column + 0}
  - \texttt{date\_column + 0}
  - \texttt{char\_column || ''}

Influencing Rule-Based Optimization

Rule-based optimization is highly sensitive to syntax. When you join tables, the order in which the tables are specified in the \texttt{FROM} clause can have a significant performance impact. For example, you should always put the most restrictive table at the end, because the optimizer evaluates the \texttt{FROM} clause from right to left. Also, the order of the predicates in a compound \texttt{WHERE} clause can influence performance.

The optimizer always uses indexes when they are available. Thus it is important to create appropriate indexes and drop indexes that deteriorate performance. Unfortunately, most of the unique indexes cannot be dropped because they protect integrity constraints.

The standard way to prevent rule-based optimization from using existing indexes is to use a trick: Indexes are usable only when the associated column names appear clean in the predicates. By applying a dummy operator (adding zero or concatenating an empty string), you suppress the index usage. You will investigate this optimizer behavior in more detail in the workshops.

\textbf{Note:} Cost-based optimization provides a more flexible, readable, and acceptable method to influence the optimizer. Modifying predicates in the way described above still works, but is not recommended. However, you should understand RBO behavior when maintaining or upgrading existing applications.
Summary

In this lesson, you should have learned:

- Rule-based optimization (RBO)
- Cost-based optimization (CBO)
- Cost factors considered by CBO
- Setting the optimizer approach
- RBO ranking scheme
- Influencing rule-based optimization

This lesson introduced both rule-based and cost-based optimization. Rule-based optimization is still supported for backward compatibility. You have seen how the RBO uses a ranking scheme, and how you can influence its behavior.

Cost-based optimization was introduced in Oracle7. The optimizer uses statistics to calculate the cost of access paths, taking into account the number of logical I/Os, CPU usage, and network traffic.

You also learned how the optimizer approach can be set at the instance and session level, supporting four different settings:

- **CHOOSE** Default behavior: CBO when statistics are available, RBO when not
- **RULE** Forces RBO behavior
- **FIRST_ROWS** Optimizes response time for the first result
- **ALL_ROWS** Optimizes overall throughput
Practice Overview

This practice covers the following topics:

- Analyzing results with RBO
- Altering your session to use CBO

Note: By default, statistics exist on the tables in the SH schema.
Practice 5 (optional)

1. Use the `ALTER SESSION` statement to force your session to use the RULE optimizer mode.

2. Run the `dai.sql` script to drop all nonprimary key indexes on the CUSTOMERS and COUNTRIES tables.
   ```
   SQL> @dai
   on which table: customers
   SQL> @dai
   on which table: countries
   ```

3. Enable SQL*Plus AUTOTRACE to display execution plans and analyze the SQL statement shown below.
   ```
   SQL> select c.cust_last_name
       2   ,   c.cust_first_name
       3   ,   co.country_name
       4   ,   co.country_region
       5 from customers c
       6   ,   countries co
       7 where c.country_id = co.country_id
       8 and co.country_subregion = 'Africa';
   ```

4. Analyze the SQL statement shown below to see which optimizer mode is used.
   ```
   SQL> SELECT *
       2 FROM customers
       3 WHERE cust_credit_limit = 15000;
   ```

5. Use the `ALTER SESSION` statement to force your session to use the CHOOSE optimizer mode. Remember to turn SQL*Plus AUTOTRACE off prior to the ALTER SESSION command.

6. Turn SQL*Plus AUTOTRACE on. Re-execute the query in step 3 and examine the results. Are the results any different?

Instructor Note

This practice session is optional. Use it when you want a break between the initial (theoretical) lessons. Ideally, you should try to go through the first six lessons as quickly as possible, so that the first workshop can be started. During the three days of this course, the theory/practice ratio changes considerably.
Indexes and Basic Access Methods

Schedule: Timing    Topic
                  45 minutes  Lecture
                  45 minutes  Total

Instructor Note

Spend some time on constraints and implicit indexes. With every new release, the optimizer becomes more intelligent in its use and reuse of existing indexes. Instead of relying on unique constraints to create unique indexes as a side effect, you should manually create all indexes needed for queries before constraints are enabled.

Note that indexes will be revisited in the “Advanced Indexes” lesson; you can point this out to the students. This lesson is a short one, so that students can begin the first workshops as soon as possible.
Objectives

After completing this lesson, you should be able to do the following:

- Identify ROWIDs
- Identify index types
- Show the relationship between indexes and constraints
- Create indexes manually
- Identify index issues with foreign keys
- Identify basic access methods
- Skip scan of indexes
- Monitor index usage
- Describe an overview of clusters

Instructor Note (for page 6-3)

Index-organized tables have logical ROWIDs instead of real ROWIDs. This is discussed in the “Alternative Storage Techniques” lesson.

Demo scripts: demo06_01.sql and demo06_02.sql

Purpose: To show the students ROWIDs.
ROWIDs

- ROWIDs indicate the address of a row.
- Extended and restricted ROWIDs are available.
- Each table has a ROWID pseudocolumn.

Oracle uses ROWIDs to store addresses of table rows, such as in indexes. The extended ROWID format was introduced with the new partitioning capabilities of Oracle 8.0. A restricted ROWID data type is still available for backward compatibility. Oracle still uses restricted ROWIDs internally, when the extended format is not required.

Each Oracle table has a pseudocolumn named ROWID. ROWIDs are not stored in the database. However, you can create tables that contain columns having the ROWID data type, although Oracle does not guarantee that the values of such columns are valid ROWIDs.

The extended ROWID has a four-piece format, using a base-64 encoding technique. In the example in the slide, following are the fields and their meaning:

- **AAABDL**: Data object number; identifies the database segment
- **AAC**: Relative file number; unique within the tablespace
- **AAAF0Z**: Data block number; relative to its data file
- **AAA-AAE**: Row numbers in the block

### ROWID Manipulation

You can use the DBMS_ROWID package to extract information from an extended ROWID or to convert a ROWID from extended format to restricted format (or vice versa). See the Oracle9i Supplied Packages Reference for more information.
Indexes

- Unique and nonunique indexes
- Composite indexes
- Index storage techniques:
  - B*-tree
    - Reverse key
    - Descending
    - Function-based
  - Bitmap
  - Domain indexes

Indexes

An index is a database object that is logically and physically independent of the table data. The Oracle9i Server may use an index to access data required by a SQL statement, or use indexes to enforce integrity constraints. You can create and drop indexes at any time. The Oracle9i Server automatically maintains indexes when the related data changes.

Index Types

Indexes can be unique or nonunique. Unique indexes guarantee that no two index entries have the same value. A composite index (also called a concatenated index) is an index that you create on multiple columns in a table (up to 32). Columns in a composite index can appear in any order, and need not be adjacent in the table.

Index Storage Techniques

For standard indexes, Oracle uses B*-tree indexes that are balanced to equalize access times. Bitmap indexes are discussed in the “Advanced Indexes” lesson.

Reverse key indexes reverse the bytes of the column values. They have benefits when used for monotonically-increasing values that can lead to skewed indexes in which older values are dropped over time. They will not be covered in this course.

Function-based indexes are also covered in the “Advanced Indexes” lesson.

Domain indexes are application-specific indexes. They are created, managed, and accessed by routines supplied by an index type. They will not be covered in this course.

Oracle9i: SQL Tuning Workshop 6-4
A Description of B*-Tree Indexes

Each B*-tree index has a root block as a starting point. Depending on the number of entries, there are one or more branch block layers. A B*-tree index also consists of a set of leaf blocks containing all values plus ROWIDs pointing to the rows in the associated data segment. The leaf blocks are connected by previous and next block pointers, so that they can be traversed from left to right and vice versa.

Indexes are always balanced and they grow from the bottom and up. In certain situations, the balancing algorithm can cause the B*-tree height to increase unnecessarily. You should reorganize your indexes regularly to guarantee optimal storage efficiency and performance. This can be done using the ALTER INDEX ... REBUILD command. Since Oracle8i it is possible to rebuild some indexes online, which does not impact the performance of DML statements.

Instructor Note

You should not go into too much detail here. The internal index algorithms are undocumented and may change (and get smarter) with each new release, without notification. Just point out what the last sentence above states: after a period of heavy DML activity, rebuilding your indexes is a good idea.
Format of Index Leaf Entries
An index entry is composed of the following components:

- An entry header, which stores the number of columns and locking information
- Key column length-value pairs, which define the column size in the key that is followed by the column value (the number of such pairs is a maximum of the number of columns in the index)
- ROWID of a row, which contains the key values

Branch block Characteristics
Branch blocks are for searching and they store the following:

- The minimum key prefix needed to make a branching decision between two keys
- The pointer to the child block that contains the key

If the blocks have $n$ keys then they have $n+1$ pointers. The number of keys and pointers is limited by the block size.

Index Leaf Entry Characteristics

- Key values are repeated if there are multiple rows that have the same key value.
- There are no index entries to rows that have all key columns containing NULL values.
- For nonpartitioned tables, restricted ROWIDs are used to point to the rows of the table, because all rows belong to the same segment.

Instructor Note
Root blocks and branch blocks are compressed by default. The leaf blocks take approximately 99% of index storage. Oracle9i also allows key compression (for concatenated indexes only) with the COMPRESS option of the CREATE INDEX or ALTER INDEX REBUILD commands. See the Oracle9i SQL Reference for more details. However, you should not offer too much technical detail here.
B*-Tree Index Example

The internal structure of a B*-tree index allows rapid access to the indexed values. The Oracle9i Server can directly access rows after it has retrieved the address (the ROWID) from the index leaf blocks.

If the optimizer decides to use an index, then the Oracle9i Server searches the index for the column values accessed by the statement and takes the following steps:

- If the statement accesses only columns that are present in the index, it retrieves the column values directly from the index (fast full index scan) without going to the data segment.
- If the statement also accesses other columns, it retrieves the addresses from the index, and then accesses the data blocks by ROWID.

Instructor Note:

Fast full index scans (also covered on page 14) have the following restrictions:

- At least one indexed column of the table must have the NOT NULL constraint.
- There must be a parallel clause on the index if you want to perform fast full index scan in parallel. The parallel degree of the index is set independently. The index does not inherit the degree of parallelism of the table.
- Make sure that you have analyzed the index; otherwise, the optimizer might decide not to use it.
CREATE INDEX Syntax

The NOSORT option can be used only with standard (B*-tree) indexes, and only if the rows in the table are exactly in the right order. It will skip the sort phase, speeding up the index creation. If the rows are not in the right order, the following error will be returned:

ORA-01409: NOSORT option may not be used; rows are not in ascending order

The NOLOGGING option reduces the undo generated during the index creation.

The physical attributes clause contains details about the physical storage for the index to be created. The details are not discussed in this course.

You can use the column expression to create function-based indexes, which are discussed in the “Advanced Indexes” lesson. In the most simple case, the column expression is just a single column name.

With the ASC and DESC options, you can specify whether the index values should be in ascending or descending order. Oracle treats descending indexes as if they were function-based indexes. You do not need the QUERY REWRITE or GLOBAL QUERY REWRITE privileges to create them, as you do with other function-based indexes. However, as with other function-based indexes, Oracle does not use descending indexes until you first analyze the index and the table on which the index is defined.

The default index is nonunique B*-tree if UNIQUE and BITMAP are not specified.

Note: This is not the full CREATE INDEX syntax; refer to the Oracle9i SQL Reference for more details.
Composite Index Guidelines

- Put the most frequently queried column first (also called the leading part of the index)
- Put the most restrictive column first
- Add extra columns to the index

When creating a composite index, use the following (possibly conflicting) guidelines:

- Put the most frequently queried column first (also called the leading part of the index), because the optimizer can use it as an index in its own right.
- Put the most restrictive column first if you expect to specify the full key.
- Consider adding extra columns to the index, so that you can retrieve query results without accessing the base table. This requires additional storage capacity but saves I/O.

When creating composite indexes you might consider using the COMPRESS option to eliminate repeated occurrences of key column values. This can substantially reduce storage. For example:

```
SQL> create index cust_name_idx
    2 on customers(cust_first_name, cust_last_name) compress 1;
```

This compresses repeated occurrences of cust_first_name values. NOCOMPRESS is the default.

**Note:** Remember that the Oracle server must maintain indexes, thus slowing down DML statements; see the next page for more details. Create only those indexes that you are sure the optimizer will use to speed up queries. It is recommended that indexes be placed in tables separate from tables. This can be done using the storage clause while creating the index. This will improve performance by reducing I/O and fragmentation.
Index Statistics

Index statistics can be collected with the following commands:

```
SQL> analyze index <index_name>
    2  validate structure;

SQL> select * from index_histogram;

SQL> select * from index_stats;
```

Index Statistics

In the “Collecting Statistics” lesson, statistics are discussed in detail. You can collect and retrieve some useful index statistics with the following commands:

```
SQL> analyze index <index_name> validate structure;
SQL> select * from index_histogram;
SQL> select * from index_stats;
```

The first command populates the two views that are queried by the last two commands. The first view gives information about the selectivity of the index (cardinality of key values), and the second view can tell you the height of the B*-tree, the number of blocks, the number of distinct key values, and the number of leaf block entries.

**ALTER INDEX Statement**

You can use the ALTER INDEX statement to change an index definition, or rebuild an existing index. On the next page you see some guidelines about when this could be needed. See the Oracle9i SQL Reference for the full ALTER INDEX statement syntax.

**Instructor Note**

Typically, a DBA uses these views to monitor and maintain statistics.
Effect of DML Operations on Indexes

- Inserts result in the insertion of an index entry in the appropriate block (Block splits might occur.)
- Deletes result in a logical deletion of the index entry (Empty blocks become available.)
- Updates to the key columns result in a logical delete and an insert to the index.

The Oracle server maintains all the indexes when DML operations are carried out on the table. Here is an explanation of the effect of a DML command on an index:

- Insert operations result in the insertion of an index entry in the appropriate block. If there is no room for the entry in the block, then it is split into two blocks, and an entry is added in the parent block to point to the new block.
- Deleting a row results only in a logical deletion of the index entry. The space used by the deleted row is not available for new entries until all the entries in the block are deleted.
- Updates to the key columns result in a logical deletion and an insertion to the index. The PCTFREE setting has no effect on the index except at the time of creation. A new entry can be added to an index block even if it has less space than that specified by PCTFREE.

After periods of high DML activity, you should reorganize your B*-tree indexes. This can be done online in Oracle8i and later. Indexes created (or rebuilt) on existing data are always more efficient than indexes that are implicitly maintained by the Oracle Server.

Instructor Note

Space released by a deleted index entry can be reused by a new entry if its value falls between the logically prior and logically subsequent values of the deleted value. A large number of deletions on an indexed table can lead to a large amount of wasted space. This can negatively impact performance during operations such as range scans of the index.
Indexes and Constraints

The Oracle server implicitly creates or uses B*-tree indexes when you define the following:

- Primary key constraints
- Unique constraints

```sql
CREATE TABLE channels
(
  channel_id CHAR(1)
  CONSTRAINT chan_pk PRIMARY KEY
, channel_desc VARCHAR2(20)
  CONSTRAINT chan_desc_nn NOT NULL
, channel_class VARCHAR2(20)
, channel_total VARCHAR2(13)
);
```

Indexes and Constraints

When you define a primary or unique key constraint on a table, the Oracle database automatically generates an index, or uses an existing index, to support it. Indexes created for this purpose are physically, but not logically, independent of the table structure.

The Oracle server gives the index the same name as the constraint.

You cannot drop an index that enforces a constraint. Instead, you must disable the constraint, in most cases causing the Oracle database to automatically drop the index.

Be aware that the Oracle database supports deferrable constraints. Unique deferrable constraints are always enforced using nonunique indexes. Also, when disabling such a constraint, the Oracle server does not drop the associated index, thus offering an easy check when the constraint is enabled again.

Indexes serve at least two purposes: to make queries faster and to enforce unique and primary keys. The Oracle optimizer uses existing indexes when new constraints are defined. Note that the Oracle database can use nonunique indexes to check unique constraints.

**Note:** Instead of relying on unique constraints to create unique indexes as a side effect, you should manually create all indexes needed for queries before constraints are enabled, including unique indexes.

For more information about this topic, see *Oracle9i Concepts* and *Oracle9i Administrator’s Guide.*
Indexes and Foreign Keys

The Oracle database does not automatically create an index for a foreign key constraint. If the foreign key columns are often used in join conditions, then you should create an index on them to enhance the join process.

In earlier releases deleting or updating data in parent-child tables (with a foreign key constraint) caused different locking behavior, depending on the presence of an index on the foreign key column. If you do not index a foreign key, the child table is locked (with a table-level lock) whenever the following occurs:

- A row is deleted from the parent table
- The columns pointed to by the foreign key are updated in the parent table

For this reason, you had to index foreign key columns, even if they are not normally used in join conditions and WHERE clauses.

In Oracle9i this locking behavior is no longer an issue. A lock is temporarily placed, and then immediately removed, before it can cause any kind of contention.

Note: For more detailed information about indexes, foreign keys, and locking behavior, see the Oracle9i Application Developer’s Guide, “Maintaining Data Integrity.”
Basic Access Methods

- **Full table scans:**
  - Can use multiblock I/O
  - Can be parallelized

- **Index scans:**
  - Allow index access only
  - Are followed by access by ROWID

- **Fast full index scans:**
  - Can use multiblock I/O
  - Can be parallelized

Full Table Scans

A full table scan may involve reading many sequential data blocks from database files on disk into the buffer cache. However, the Oracle server can read several sequential blocks in a single I/O. This is referred to as multiblock I/O. You can influence multiblock I/O by setting the `DB_FILE_MULTIBLOCK_READ_COUNT` parameter. Using the parallel query capability can help speed up full table scans by allocating several processes to the work.

Index Scans

Indexes improve the performance of many SQL statements. However, an indexed query may read from both index and data blocks. These blocks are not usually sequential, so the query cannot use multiblock reads. Use indexes only when they improve performance.

As a general rule, use B*-tree indexes for queries that select fewer than 5% of the rows from a table. However, this break-even point varies with your data. Make sure to test this for all critical processes by using diagnostic tools.

Fast Full Index Scans

The fast full index scan is an alternative to a full table scan when there is an index that contains all the columns that are needed for a query. It is faster than a normal index scan because it can use multiblock I/O and can be parallelized in the same way as a table scan. Unlike regular index scans, however, the rows will not necessarily come back in sorted order. If there is a predicate that narrows the index search to a range of values, then the optimizer will not consider a fast full index scan.
Skip Scanning of Indexes

• Index skip scanning enables access through a composite index when the prefix column is an unknown value
• Is supported by:
  – Cluster indexes
  – Descending scans
  – CONNECT BY clauses
• Is not supported by reverse key or bitmap indexes

Skip Scanning Definition

In prior releases, a composite index would be used only if the index prefix (leading) column was included in the predicate of the statement. With Oracle9i, the optimizer can use a composite index even if the prefix column value is not known. The optimizer uses an algorithm called skip scanning to retrieve ROWIDs for values that do not use the prefix column.

Skip scans reduce the need to add an index to support occasional queries that do not reference the prefix column of an existing index. This can be useful when high levels of DML activity are degraded by the existence of too many indexes used to support infrequent queries. The algorithm is also valuable in cases where there are no clear advantages as to which column to use as the prefix column in a composite index. The prefix column should be the most discriminating, but also the most frequently referenced in queries. Sometimes, these two requirements are met by two different columns in a composite index, forcing a compromise or the use of multiple indexes.
Skip Scanning Definition (continued)

During a skip scan, the B*-tree is probed for each distinct value in the prefix column. Under each prefix column value, the normal search algorithm takes over. The result is a series of searches through subsets of the index, each of which appears to result from a query using a specific value of the prefix column. However, with the skip scan, the value of the prefix column in each subset is obtained from the initial index probe rather than from the command predicate.

In addition to standard B*-tree indexes, the optimizer can use skip scans to process:

- Cluster indexes
- Descending scans
- Statements with CONNECT BY clauses

Reverse key and bitmap indexes do not support the skip scan algorithm.
Skip Scanning Example

Skip scanning is useful when the initial column of the composite index is not specified in the query. In other words, it is “skipped.”

For example, suppose you have the following index data:

(‘F’, 98)
(‘F’, 100)
(‘F’, 102)
(‘F’, 104)
(‘M’, 101)
(‘M’, 103)
(‘M’, 105)

Skip scanning is advantageous if there are few distinct values of the leading column of the composite index and many values of the nonleading key of the index.

The index is split logically in the following two subindexes:
1. The first one has the keys with the value ‘F’.
2. The second one has all the values with the value ‘M’.

If you have an EMPLOYEES table with a gender column, in the following query,

```
SELECT *
FROM employees
WHERE empno = 101;
```

the column gender is skipped. A complete scan of the index is not performed, but the subindex with the value ‘F’ is searched first, then the subindex with the value ‘M’ is searched.
Identifying Unused Indexes

- **Oracle9i** provides the capability to gather statistics about the usage of an index.
- **Benefits include:**
  - Space conservation
  - Improved performance by eliminating unnecessary overhead during DML operations

---

Identifying Unused Indexes

Oracle9i provides the capability to gather statistics within the database about the usage of an index.

**Benefits**

If you find an index that is never used, you can drop the index and free the space it used.

Indexes must be maintained during most inserts, deletes, and some updates. Eliminating an unused index eliminates this overhead.

Parsing is simplified when the optimizer has fewer indexes to consider.
Enabling and Disabling the Monitoring of Index Usage

• To start monitoring the usage of an index:

   ALTER INDEX customers_pk MONITORING USAGE;

• To stop monitoring the usage of an index:

   ALTER INDEX customers_pk NOMONITORING USAGE;

• V$OBJECT_USAGE and V$OBJECT_STATS contain information about the usage of an index.

Enabling and Disabling the Monitoring of Index Usage

The V$OBJECT_USAGE view displays information about the usage of an index. Each time the index is altered with the MONITORING USAGE clause, V$OBJECT_USAGE is reset for the specified index. Previous information is lost, and a new start time is recorded.

The new V$OBJECT_USAGE view supports the identification of unused indexes with the following columns:

- INDEX_NAME: The index name
- TABLE_NAME: The corresponding table
- MONITORING: Indicates whether monitoring is on or off
- USED: Indicates whether an index has been used during the monitoring time
- START_MONITORING: Time monitoring began on an index
- STOP_MONITORING: Time monitoring stopped on an index
Clusters

- Rows are grouped and physically stored together.
- Tables in a cluster are prejoined.
- Clusters are transparent to users and applications.
- DML and full table scans are more expensive.

Clusters: Overview

A cluster is a structure that is used to physically locate rows together because they share common column values. There are two type of clusters: index clusters and hash clusters.

The columns used to cluster the rows are called the cluster key:

- The cluster key can consist of one or more columns.
- Tables in a cluster have columns that correspond to the cluster key. Clustering is a mechanism that is transparent to the applications using the tables. Data in a clustered table can be manipulated as though it were stored in a regular table.
- Updating one of the columns in the cluster key may cause the Oracle server to physically relocate the row.
- The cluster key is independent of the primary key. The tables in a cluster can have a primary key, which may be the cluster key or a different set of columns.

Clusters can waste disk space if the cluster key distribution is skewed. Choosing an appropriate cluster size is one of the most important issues when you consider whether to cluster your tables.
Cluster Example

If they are stored as regular tables, CUSTOMERS and COUNTRIES are placed in different segments. This means that the tables use their own set of blocks—a block that stores rows from the CUSTOMERS table does not contain data from the COUNTRIES table, and vice versa.

Because customers are usually accessed by COUNTRY_ID, the developer may cluster both tables or just the CUSTOMERS table on the COUNTRY_ID column. If the tables CUSTOMERS and COUNTRIES are stored in a cluster, they share the same cluster segment; see the example in the slide. A block in this segment stores rows from both tables. This means that a full table scan on the COUNTRIES table takes more time, because its rows are interspersed with the rows from the CUSTOMERS table.

If a table is stored in a cluster, the cluster becomes the physical unit of storage, and the table is a logical entity; that is, the clustering is transparent to the users and applications.

Instructor Note

These two tables are not ideal for clustering; the slide is meant only for illustration purposes. Note that this is an index cluster.
Index Clusters

An index cluster uses a special index, known as the cluster index, to maintain the data within the cluster. In the example on the previous page, when a user inserts a new customer, the cluster index ensures that the new customer is placed in the same block as the other customers with the same `country_id`.

Choose Appropriate Columns for the Cluster Key

Choose cluster key columns carefully. If you use multiple columns in queries that join the tables, make the cluster key a composite key. In general, the same column characteristics that make a good index apply to cluster indexes.

A good cluster key has enough unique values so that the group of rows corresponding to each key value fills approximately one or two data blocks. Too many rows per cluster key value can require extra searching to find rows for that key. Cluster key values that are too general (such as `MALE` and `FEMALE`) result in excessive searching and can result in poor performance.

Instructor Note

The “Choose Appropriate Columns for the Cluster Key” paragraph is also true for hash clusters.
Index Clusters: Performance Characteristics

- Can save disk space
- Can reduce disk I/O
- Improve range scan access time
- Improve join performance for clustered tables

Index Clusters: Performance Characteristics

Index clusters can save disk space, because each cluster key value is stored only once for all the rows that have the same key value.

When accessing rows with a common value, disk I/O is reduced and access time improves. Tables that are often joined using a foreign key relationship are good choices for an index cluster. Because rows with a given key value are stored together, using a cluster can reduce the number of blocks read to satisfy such a request.

In some situations, it is useful to place only the detail table in a cluster. Consider a case where customer information is always accessed by CUST_ID, and full table scans of the CUSTOMERS table are frequent. In this case, placing only the COUNTRIES table in an index cluster may be the best solution, thus the full table scans of the CUSTOMERS table do not suffer any performance degradation.

The cluster index must be available to store, access, or maintain data in an index cluster. The cluster index points to the block that contains the rows with a given key value. The structure of a cluster index is similar to that of a normal index. Although a normal index does not store NULL key values, cluster indexes do store NULL keys. There is only one index entry for each key value in the cluster index. Therefore, cluster indexes are likely to be smaller than a normal B*-tree index on the same set of data.

To store or retrieve rows from a cluster, the Oracle server uses the cluster index to locate the first row that corresponds to the given key value and then retrieves the rows for the given key.
Index Clusters: Limitations and Guidelines

- DML does not perform as well.
- Avoid updating cluster key values.
- Index clusters are more suitable when the occurrence of the key value is uniform.
- Index clusters are less suitable where large amounts of data are being loaded.

DML Performance
DML does not perform as well on a cluster as it does with a table. When inserting a row into a cluster, it must be inserted into the same block with the other rows with the same cluster key. If several rows with different cluster keys are being inserted, then several blocks must be read. However, for a regular table, the rows can be inserted into the single block that is the first block on the free list.

Cluster Key Column Updates
When the key value in a cluster is updated, the row may need to be moved physically to another block. This can cause degradation in performance. Therefore, columns that are frequently updated are not good candidates for being part of the cluster key.

Row Frequency
Clusters are generally more suitable in situations where the frequency of occurrence of key values is uniform. In the COUNTRIES example, if the countries contain almost an equal number of customers, then clustering may be useful. If a certain country has only one or two customers and others have several hundred, clustering may not be justified. If a country has many customers, then some of the customer rows may need to be stored in overflow blocks.

Direct Loads
Clusters cannot be loaded using direct loads.
Hash Clusters

A hash cluster uses a hash function to calculate the location of a row. The hash function uses the cluster key and can either be system generated or defined by the developer.

When a row is inserted into a table in a hash cluster:
- The hash key columns are used to compute a hash value
- The row is stored based on the hash value

The hash function is also used to find the row while retrieving the data from a hashed table.

**Example**

Customer rows are stored in a hash cluster using the Customer ID as the cluster key. When Customer 501 is inserted into the cluster, the hash function determines where to store the row using the cluster key value 501. Later, when Customer 501 is needed for retrieval, the cluster key value 501 is input into the hash function, which points to the location where the row was inserted.

**Hash Cluster Performance Characteristics**

The hash function is used to locate the row while retrieving the data from a hashed table without the I/O and CPU time required to search through an index. Because large tables have indexes with more B*-tree levels, hash clusters perform better when there are many rows in the table.

*Oracle9i: SQL Tuning Workshop 6-25*
Hash Clusters: Limitations

- Range scans cannot use the hash function.
- Unevenly distributed cluster key values can cause overflow, which implies additional I/O.
- Volatile and unpredictable data can result in an inefficient hash function.
- Full table scans are generally slower on clustered tables.

Hash Clusters: Limitations

The hash function can be used to retrieve data only if the key value is known. For this reason, a range search cannot use the hash function. If queries use equality predicates on the cluster key, they can benefit from hashing. However, it is possible to create an index on the hash cluster key if many queries use range searches on the hash cluster key columns.

If cluster keys are not evenly distributed throughout the hash cluster, then many of them hash into the same blocks. When these blocks fill, rows get stored into overflow blocks. It takes additional I/Os to access these rows.

In a hash cluster, the hash function is based on the parameters specified by the developer at the time of its creation. If it is difficult to predict the key values or their size accurately because of the volatility of the data, a wrong choice could result in an inefficient hash function, causing either a lot of overflow or a waste of space. Therefore, hashing is useful only if the number of rows per cluster key and the number of cluster keys are predictable.

The hashing algorithm uses the number of distinct cluster keys that will be stored in the cluster when it determines where to store the row. There must be enough space allocated to the cluster to store all of these rows. Because rows can be stored anywhere in the cluster, the Oracle server reads the entire cluster when performing a full table scan, even though there may be few rows in the cluster. Contrast this with a table that is loaded from the front to the back. The Oracle server only needs to read to the block that stores the last row in the table. Therefore, full table scans are generally slower on clustered tables than on nonclustered tables.
When to Use Clusters

- **Use clusters:**
  - With tables that are primarily queried or joined on the cluster key
  - To even cluster key distribution
- **Clusters can also reduce performance:**
  - DML statements
  - Full table scans

When to Use Clusters

Use clusters to store one or more tables that are queried primarily on the cluster key (not predominantly inserted or updated), and for which queries often join data of multiple tables in the cluster or retrieve related data from a single table.

All of the rows for a single cluster key can fit into one or two Oracle blocks. If not, the Oracle server must read multiple blocks when querying using the cluster key. To identify data that would be better stored in clustered form than nonclustered, look for tables that are related through referential integrity constraints and for tables that are frequently accessed together using queries that join data from two or more tables.

If you cluster tables on the columns used to join table data, you reduce the number of data blocks that must be accessed to process the query; all the rows needed for a join on a cluster key are in the same block. Therefore, query performance for joins is improved.

Like indexes, clusters do not affect application design. The existence of a cluster is transparent to users and applications. Data stored in a clustered table is accessed through SQL just like data stored in a nonclustered table. Clusters can reduce the performance of:

- DML statements (**INSERT**, **UPDATE**, and **DELETE**) (Because of the overhead of maintaining multiple tables in a single block, DML against a table is more efficient than DML against a cluster.)
- Full table scans (When performing a full table scan on a cluster with multiple tables, all of the tables are read, because rows from the tables are stored in the same blocks.)
When to Use Clusters

- **When to use index clusters:**
  - Appropriate physical data structure
  - Hash cluster cannot be used
- **When to use hash clusters:**
  - Query performance for equality searches is the most important goal
  - Tables are large

When to Use Index Clusters

Consider using index clusters in those situations where the cluster is the appropriate physical data structure, but a hash cluster cannot be used because of its limitations.

When to Use Hash Clusters

Consider hash clusters in these situations:

- `SELECT` performance is the primary tuning goal.
- The rows in the hash cluster are primarily queried with the equality (=) operator.
- The table is large, so a B*-tree index would become too high. Accessing rows through a hash function bypasses index I/O. With a large table, a hash cluster can be chosen as the data structure, even if the rows do not need to be clustered. In this situation, the hash cluster is used because it eliminates index I/Os, not because it clusters rows.
- Cluster keys are evenly distributed throughout the hash cluster.

Do not use hash clustering when you access data often without an equality comparison on the cluster key:

- Range scans cannot use the hashing algorithm. Thus, this type of access causes full table scans. However, if range scans are required, the developer can create an additional index.
- Full table scans may perform poorly because hash clusters tend to have a lot of unused space that is read during full table scans.
Summary

In this lesson, you should have learned about the following:

- ROWIDs
- Indexes
  - Index types
  - B*-tree index structure
  - DML operations and indexes
  - Indexes and constraints
- The CREATE INDEX syntax

Summary

This lesson introduced you to indexes. An index is a database object that is logically and physically independent of the table data. The Oracle server may use an index to access data required by a SQL statement, or use indexes to enforce integrity constraints.

There are different index types. For standard indexes, Oracle uses B*-tree indexes that are balanced to equalize access times. You can create an index manually by using the CREATE INDEX syntax, or automatically by creating unique or primary key constraints on tables.

Indexes can impact performance. There are different methods to scan a table: full table scans, index scans, and fast full index scans. Index scans can improve the performance of many SQL statements.
Summary

• Basic access methods
  – Full table scans
  – B*-tree index scans
  – Access by ROWID
  – Fast full index scans
  – Skip scan of indexes
  – Index usage monitoring
  – Overview of clusters

Instructor Note

After completing this lesson, it is possible to do workshops 1 (single table, single predicate) and 2 (sorting, grouping, and set operators). Note that the next lesson has a short practice session. Thus, a typical approach might be to do workshop 1 now, continue with the next two lessons, and then do workshop 2. But, as indicated, you can do workshop 2 earlier as well.
Collecting Statistics

Schedule: Timing     Topic
   45 minutes    Lecture
   20 minutes    Practice
   65 minutes    Total

Instructor Note

The main topics in this lesson are statistics, how to collect statistics with the `ANALYZE` command, using `DBMS_STATS`, predicate selectivity, and histograms. The short practice session reviews collecting and viewing some statistics. There is a new utility called `statspack` which is similar to `utlbstat/utlestat` scripts, which collect all system and application statistics. This is not covered in this course.
Objectives

After completing this lesson, you should be able to do the following:

• Use the `ANALYZE` command to collect statistics
• Identify table, index, column, and cluster statistics
• Identify predicate selectivity calculations
• Use the `DBMS_STATS` package
• Create histograms for columns with skewed data
The **ANALYZE** Command

You collect statistics on an object with the **ANALYZE** command. Although the optimizer is not sensitive to minor changes in volume or selectivity, you may want to collect new statistics periodically on frequently modified tables to ensure that the optimizer is using recent, accurate information.

Using the **ANALYZE** command to collect new statistics overwrites any existing statistics in the data dictionary and flushes any related execution plans from the shared pool.

The **DBMS_DDL** package contains a procedure called **ANALYZE_OBJECT** that allows an index, table, or cluster to be analyzed. The **DBMSUTILITY** package contains a procedure called **ANALYZE_SCHEMA** that gathers statistics on all objects in the given schema.

**COMPUTE STATISTICS** Option

You calculate exact statistics with this option. It performs a full table scan and several calculations. For large tables, this operation could take a considerable amount of time.

**ESTIMATE STATISTICS** Option

You estimate statistics with this option. However, if you use this option with a suitable sample of the data, it is almost as reliable as the **COMPUTE STATISTICS** option.

**DELETE STATISTICS** Option

You clear out statistics with this option. You do not need to use this option before reanalyzing an object, because existing statistics are overwritten.

---

*Oracle9i: SQL Tuning Workshop 7-3*
The **ANALYZE Command**: Sampling

If more than half the data is specified as a sample size, the Oracle9i Server reads all the data and computes the statistics rather than estimating them. If the table has fewer than 1,064 rows and a sample size is not specified, the Oracle9i Server will compute the statistics regardless of whether \texttt{ESTIMATE STATISTICS} or \texttt{COMPUTE STATISTICS} is chosen.

**Note:** The default \texttt{ANALYZE} behavior is the combination of \texttt{FOR TABLE} and \texttt{FOR ALL INDEXES}.

**FOR Clause**

The \texttt{FOR} clause is meant for histograms and will be discussed in more detail later in this lesson.
Table Statistics

- Number of rows
- Number of blocks (*always exact*)
- Number of empty blocks (*always exact*)
- Average available free space
- Number of chained or migrated rows
- Average row length
- Last ANALYZE date and sample size
- Data dictionary views:
  - USER_TABLES
  - ALL_TABLES

Table Statistics

Following are the table statistics collected by the ANALYZE command:

- Number of rows
- Number of blocks (always exact)
- Number of empty (never used) blocks (always exact)
- Average row length (in bytes)
- Average available free space (in bytes per block)
- Number of chained or migrated rows
- Last ANALYZE date and sample size

**Note:** Chaining occurs when a single row is too big to fit in any database block, usually because column widths are too big, or there are too many columns defined for the table.

Migration happens when the space reserved in the block for updates is insufficient, which causes the row being updated to be moved into another block, leaving a pointer in the original block to the new address for the row.
ALL_TABLES and DBA_TABLES Data Dictionary Views

The table statistics are stored in the data dictionary. They can be displayed by querying the USER_TABLES or ALL_TABLES view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>TABLESPACE_NAME</td>
<td></td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>NUM_ROWS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>BLOCKS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>EMPTY_BLOCKS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>AVG_SPACE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>CHAIN_CNT</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>AVG_ROW_LEN</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>SAMPLE_SIZE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>LAST_ANALYZED</td>
<td></td>
<td>DATE</td>
</tr>
</tbody>
</table>

Example

```
SQL> analyze table products
2  estimate statistics for table;

SQL> select num_rows, blocks, empty_blocks
2  , avg_space, avg_row_len
3  , sample_size
4  from user_tables
5  where table_name = 'PRODUCTS';
```

```
NUM_ROWS  BLOCKS EMPTY_BLOCKS AVG_SPACE AVG_ROW_LEN SAMPLE_SIZE
---------- ---------- ------------ ---------- ----------- -----------
10000    714       53        292         260       1007
```

Instructor Note

Demonstration: First delete all non-primary key indexes from the PRODUCTS table by running the dai.sql script.

Second, run the demo07_01.sql script to demonstrate the code shown.

Note that the SAMPLE_SIZE column contains a number that could indicate a number of rows or a percentage, depending on the last sample type used. Also note that the NUM_ROWS and BLOCKS values are the maximum and minimum value for the clustering factor of any index created on the table. The clustering factor is introduced on the next page.
Index Statistics

- Index level (*always exact*)
- Number of leaf blocks
- Number of distinct keys
- Average number of leaf blocks per key
- Average number of data blocks per key
- Number of index entries
- Clustering factor
- Last ANALYZE date and sample size

Index Statistics

Note that the index B*-tree-level statistic is always exact, regardless whether you compute or estimate index statistics. The number of distinct keys may include rows that have been deleted.

Index Clustering Factor

The index clustering factor is an important index statistic for the CBO to estimate index scan costs. It is an indication of the number of (logical) data block visits needed to retrieve all table rows via the index. If the index entries follow the table row order, this value approaches the number of data blocks (each block is visited only once). Conversely, if the index entries randomly point at different data blocks, the clustering factor could approach the number of rows.

Suppose a typical table contains 50 rows per data block and 5,000 rows in total. When a query looks for 50 rows (selectivity is 1%), the clustering factor indicates whether you must visit one block (best case: 1% of the data blocks) or 50 blocks (worst case: 50% of the data blocks).

Instructor Note

The index clustering factor is a measure of how well ordered the data is in the table in reference to the key.
Index Statistics

- Collect index statistics while creating or rebuilding indexes
- Data dictionary views:
  - USER_INDEXES
  - ALL_INDEXES

```sql
SQL> select uniqueness, blevel
2    , leaf_blocks, distinct_keys
3    , clustering_factor
4 from user_indexes
5 where index_name = 'PRODUCTS_PK';
```

Index Statistics (continued)

You can collect index statistics with the ANALYZE command, but you can also collect index statistics while creating or rebuilding them by using the following syntax:

- `SQL> create index ... compute statistics`
- `SQL> alter index ... rebuild compute statistics`
**USER_INDEXES and ALL_INDEXES Data Dictionary Views**

The index statistics are stored in the data dictionary. They can be displayed by querying the USER_INDEXES or ALL_INDEXES view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>INDEX_TYPE</td>
<td></td>
<td>VARCHAR2(12)</td>
</tr>
<tr>
<td>UNIQUENESS</td>
<td></td>
<td>VARCHAR2(9)</td>
</tr>
<tr>
<td>BLEVEL</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>LEAF_BLOCKS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>DISTINCT_KEYS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>AVG_LEAF_BLOCKS_PER_KEY</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>AVG_DATA_BLOCKS_PER_KEY</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>CLUSTERING_FACTOR</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>NUM_ROWS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>SAMPLE_SIZE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>LAST_ANALYZED</td>
<td></td>
<td>DATE</td>
</tr>
</tbody>
</table>

**Example**

```sql
SQL> analyze index products_pk compute statistics;

SQL> select uniqueness, blevel
2  , leaf_blocks, distinct_keys
3  , clustering_factor
4  from user_indexes
5  where index_name = 'PRODUCTS_PK';
```

<table>
<thead>
<tr>
<th>UNIQUENES</th>
<th>BLEVEL</th>
<th>LEAF_BLOCKS</th>
<th>DISTINCT_KEYS</th>
<th>CLUSTERING_FACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIQUE</td>
<td>1</td>
<td>43</td>
<td>10000</td>
<td>709</td>
</tr>
</tbody>
</table>

**Instructor Note**

Demo: Run the demo07_02.sql script to analyze an index and display information about the analyzed index. You can analyze the PRODUCTS_PK as shown in the example, or choose another index to analyze.
Column Statistics

- Number of distinct values
- Lowest value
- Highest value
- Last ANALYZE date and sample size
- Data dictionary views:
  - USER_TAB_COL_STATISTICS
  - ALL_TAB_COL_STATISTICS

Instructor Note
This statement causes histograms to be created for every column. This is not advisable.

SQL> analyze table products
  2  compute statistics for all columns;
Instead, use the syntax FOR TABLE FOR COLUMNS, such as:
SQL> analyze table products compute statistics
  2  for table for columns prod_list_price
  3  size 50;
(This example is shown on page 7-22.)
Both the lowest and the highest value are stored in RAW (binary) format.
USER/ALL_TAB_COL_STATISTICS Data Dictionary Views

The column statistics are stored in the data dictionary. They can be displayed by querying the USER_TAB_COL_STATISTICS or ALL_TAB_COL_STATISTICS view.

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>COLUMN_NAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>NUM_DISTINCT</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>LOW_VALUE</td>
<td></td>
<td>RAW(32)</td>
</tr>
<tr>
<td>HIGH_VALUE</td>
<td></td>
<td>RAW(32)</td>
</tr>
<tr>
<td>DENSITY</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>NUM_NULLS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>NUM_BUCKETS</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>LAST_ANALYZED</td>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td>SAMPLE_SIZE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>GLOBAL_STATS</td>
<td></td>
<td>VARCHAR2(3)</td>
</tr>
<tr>
<td>USER_STATS</td>
<td></td>
<td>VARCHAR2(3)</td>
</tr>
<tr>
<td>AVG_COL_LEN</td>
<td></td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

Note: For backward compatibility with Oracle7, the USER/ALL_TAB_COLUMNS data dictionary views also show column statistic values. Also, the NUM_BUCKETS column shows that regular column statistics are treated as a histogram with one bucket. Histograms are discussed later in this lesson.

Example

```sql
SQL> select column_name, num_distinct
2 , low_value, high_value
3 , num_nulls, num_buckets
4 from user_tab_col_statistics
5 where table_name = 'PRODUCTS';
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>DISTINCT</th>
<th>LOW_VALUE</th>
<th>HIGH_VALUE</th>
<th>NULLS</th>
<th>BUCKETS</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROD_ID</td>
<td>10000</td>
<td>C106</td>
<td>C306</td>
<td>0</td>
<td>75</td>
</tr>
<tr>
<td>PROD_NAME</td>
<td>759</td>
<td>312D506965</td>
<td>5A69702D46</td>
<td>0</td>
<td>75</td>
</tr>
<tr>
<td>PROD_DESC</td>
<td>671</td>
<td>7468697320</td>
<td>7468697320</td>
<td>0</td>
<td>75</td>
</tr>
<tr>
<td>PROD_SUBCATEGORY</td>
<td>37</td>
<td>4361737561</td>
<td>556E646572</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>PROD_SUBCAT_DESC</td>
<td>31</td>
<td>7468697320</td>
<td>7468697320</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>PROD_CATEGORY</td>
<td>4</td>
<td>426F7973</td>
<td>576F6D565E</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

... 14 rows selected.

Instructor Note

Demo script: demo07_03.sql

If you have time, you can show the BOIL_RAW function here. You create this function by running the boilraw.sql script. This is an unsupported script. The function is used by the colstats.sql script, but you can also use the function directly; for example, see the demo07_04.sql script, and enter PROD_ID when prompted.

Note that Oracle9i also provides a DBMS_STATS.CONVERT_RAW_VALUE procedure that has the same functionality. The DBMS_STATS package is discussed later this lesson.
The `DBMS_STATS` Package

You can use the `DBMS_STATS` supplied package to generate and manage statistics for cost-based optimization. The statistics can reside in the data dictionary or in a table created in the user’s schema. Only statistics stored in the dictionary impact the cost-based optimizer.

The procedures in the package can be divided into three main functions:

- To set or get individual statistics
- To transfer statistics between the data dictionary and user tables
- To gather certain classes of optimizer statistics

**Note:** The procedures of the third category have improved (or provide equivalent) performance characteristics compared to the `ANALYZE` command.

**Reference**

For full details of the `DBMS_STATS` package, see the *Oracle9i Supplied PL/SQL Packages and Types Reference*. 
DBMS_STATS: Generating Statistics

```sql
dbms_stats.GATHER_TABLE_STATS
('SH'              -- schema
,'CUSTOMERS'       -- table
,NULL             -- partition
,20               -- sample size(%) 
,FALSE            -- block sample?
,'FOR ALL COLUMNS' -- column spec
,4                -- degree of //
,'DEFAULT'        -- granularity
,TRUE             -- cascade to indexes
);
```

DBMS_STATS Procedures

Use the following procedures to gather statistics on indexes, tables, columns, and partitions:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATHER_INDEX_STATS</td>
<td>Collects index statistics</td>
</tr>
<tr>
<td>GATHER_TABLE_STATS</td>
<td>Collects table, column, and index statistics</td>
</tr>
<tr>
<td>GATHER_SCHEMA_STATS</td>
<td>Collects statistics for all objects in a schema</td>
</tr>
<tr>
<td>GATHER_DATABASE_STATS</td>
<td>Collects statistics for all objects in a database</td>
</tr>
</tbody>
</table>

DBMS_STATS does not gather cluster statistics, but you can use it to gather statistics on the individual tables in the cluster. Other options for gathering statistics with DBMS_STATS include the following:

- Collect statistics either serially or in parallel. Index statistics are gathered only serially.
- Compute or estimate statistics; use block samples or row samples to estimate statistics.
- Specify the columns for which statistics are needed with GATHER_TABLE_STATS.
- Use the CASCADE option to collect index statistics when generating table statistics.
- Hold statistics in statistics tables to experiment with different sets of statistics.

Instructor Note

Run the demo07_05.sql script to show the above code. This demonstration script works with scripts demo07_06.sql and demo07_07.sql, which are run on the next pages.
Copy Statistics Between Databases

You can use the DBMS_STATS package to copy statistics from a production to a test database to facilitate tuning. For example, to copy a schema’s statistics, perform the following steps:

1. Use the DBMS_STATS.CREATE_STAT_TABLE procedure in the production database to create a user-defined statistics table.
2. Use the DBMS_STATS.EXPORT_SCHEMA_STATS procedure in the production database to copy statistics from the data dictionary to the user-defined statistics table from Step 1.
3. Use the export and import utilities to transfer the statistics to a corresponding user-defined statistics table in the test database.
4. Use the DBMS_STATS.IMPORT_SCHEMA_STATS procedure to import the statistics into the data dictionary in the test database.

You can also use DBMS_STATS to back up statistics prior to analyzing objects. Use a backup to:

- Restore old statistics
- Study changes in data characteristics over time
### Example: Copying Statistics

```sql
dbms_stats.EXPORT_TABLE_STATS
('SH'  -- schema
,'CUSTOMERS'  -- table name
,NULL        -- no partitions
,'STATS'      -- statistics table name
,NULL        -- id for statistics
,TRUE        -- index statistics
);
```

```sql
dbms_stats.CREATE_STAT_TABLE
('SH'       -- schema
,'STATS'    -- statistics table name
,'DATA01'  -- tablespace
);
```

To copy statistics for a table from the data dictionary to a user-defined statistics table, use the `CREATE_STAT_TABLE` procedure to create the user-defined table and `EXPORT_TABLE_STATS` to copy the statistics. `DBMS_STATS` includes the following procedures for copying statistics:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CREATE_STAT_TABLE</code></td>
<td>Creates a user-defined table capable of holding statistics</td>
</tr>
<tr>
<td><code>DROP_STAT_TABLE</code></td>
<td>Drops a user-defined statistics table</td>
</tr>
<tr>
<td><code>EXPORT_object_STATS</code></td>
<td>Exports statistics from the data dictionary to a user-defined table</td>
</tr>
<tr>
<td><code>IMPORT_object_STATS</code></td>
<td>Imports statistics from a user-defined table to the data dictionary</td>
</tr>
</tbody>
</table>

where `object` can be `COLUMN`, `INDEX`, `TABLE`, or `SCHEMA`.

### Instructor Note

Demonstration script: `demo07_06.sql`

You may need to change the name of the tablespace to your environment's tablespace name.
Example: Gathering Statistics

begin
  dbms_stats.CREATE_STAT_TABLE
  (‘SH’, ‘STATS’);
  dbms_stats.GATHER_TABLE_STATS
  (‘SH’, ‘CUSTOMERS’,
   stattab => ‘STATS’);
end;

begin
  dbms_stats.DELETE_TABLE_STATS
  (‘SH’, ‘CUSTOMERS’);
  dbms_stats.IMPORT_TABLE_STATS
  (‘SH’, ‘CUSTOMERS’,
   stattab => ‘STATS’);
end;

Gathering Optimizer Statistics

The GATHER_TABLE_STATS procedure gathers table and column (and index) statistics. There are several parameters for the procedure; most are optional and have a default value. See the Oracle9i Supplied PL/SQL Packages and Types Reference for more information.

Example

There has been a lot of modification against the CUSTOMERS table since the last time statistics were gathered. To ensure that the cost-based optimizer is still selecting the best plan, statistics are gathered again using GATHER_TABLE_STATS. However, you are concerned that new statistics will cause the optimizer to choose bad plans when the current ones are acceptable.

If you believe that the new statistics cause the optimizer to generate poor plans, you can restore the original statistics using DELETE_TABLE_STATS and SET_TABLE_STATS.

Instructor Note

Demonstration script: demo07_07.sql.
This is a second example of the GATHER_TABLE_STATS procedure; note that this time you use a user-defined statistics table. With the IMPORT_TABLE_STATS, you restore statistics from the user-defined statistics table to the data dictionary.
Predicate Selectivity

- Unique or primary key = constant
  single-row predicate
- Nonunique index = constant
  \[
  \text{selectivity} = \frac{1}{\text{distinct_keys}}
  \]
- Bounded or unbounded range scan
  \[
  \text{selectivity} = \frac{\text{high} - \text{low} + 1}{\text{max} - \text{min} + 1}
  \]
  - high: upper bound (or max)
  - low: lower bound (or min)
  - max, min: column statistics

Predicate Selectivity

The expected percentage of rows (the selectivity) depends on the sort of operation performed in the WHERE clause. The cost-based approach is more likely to choose an index access path for a query with good selectivity. The Oracle9i optimizer uses the following rules when calculating selectivity.

Equality Conditions

- Unique or primary key = constant
  This is a single-row predicate, returning one row maximally; it has maximum selectivity.

- Nonunique index = constant
  The query may return more than one row, so the optimizer uses the number of distinct key values in the column. The selectivity is 1 divided by the number of distinct values.

Bounded and Unbounded Ranges

Bounded and unbounded ranges need statistics to calculate selectivity, thus CBO and RBO yield different results. The RBO must rely on the syntax and its ranking scheme; the CBO uses the following formula:

\[
\text{selectivity} = \frac{\text{high} - \text{low} + 1}{\text{max} - \text{min} + 1}
\]

Oracle9i: SQL Tuning Workshop 7-17
Bind Variables and Predicate Selectivity

- Equality condition: No difference
- Bounded or unbounded range scan:
  Built-in default selectivity assumed
- If high performance is an issue, consider using
dynamic SQL instead of bind variables:
  - DBMS_SQL package
  - EXECUTE IMMEDIATE
- Dynamic SQL implies less shared SQL

In case of equality conditions (that is, if the operator is an equality sign), a bind variable is
treated as a constant. See the first two bullets in the previous slide.

Because the bind phase follows the parse phase, the Oracle9i optimizer does not know the actual
bind variable values when optimizing a SQL statement. This means that the optimizer must use a
built-in default selectivity in case a SQL statement contains bind variables. These built-in values
are not documented, cannot be influenced, and can change with each new release of the Oracle
server.
Performance

If performance is an issue, consider using dynamic SQL to build statements with literal values instead of bind variables. This means that each statement will be optimized separately (giving optimal performance). However, there is one big disadvantage: because most dynamic SQL statements are different from one another, you will have less benefit from shared SQL.

You can send dynamic SQL (that is, SQL statements that are built at run time) to the Oracle8i Server by using the DBMS_SQL package, or by using the EXECUTE IMMEDIATE statement. Dynamic SQL is not discussed in this class; see the PL/SQL User’s Guide and Reference for more details.

Instructor Note

Be careful to talk about built-in default selectivities. As previously stated, they are undocumented and can change without notice. Note that these default values are also used when CBO is forced and no statistics are available. Since Oracle 7.3, the default selectivity for an inequality is 5%.

Instructor Note (for page 7-20)

This slide is just an example; histograms are only useful on large sets of values. That is why it is useless to give selectivity calculations based on this example. You can use the histdemo.sql demonstration script at the end of this lesson, based on this example.
The optimizer must estimate the number of rows processed by a given query. This is achieved partly by estimating the selectivities of the query’s predicates. The accuracy of these estimates depends on the optimizer’s knowledge of the data distribution. Without histograms, an even distribution is assumed.

**Types of Histograms**

- **Width-balanced histograms** partition the attribute domain into equal width ranges, called buckets, and count the number of rows, the value of which in a column makes them fall into each bucket. If several rows contain the same value, they are all put into the same bucket, increasing the height of that bucket.

- **Height-balanced histograms** have each bucket containing approximately the same number of rows. If several rows contain the same value, they may be put in the same bucket or spread across several buckets, as the histogram balances the heights of the buckets. Some buckets cover only one or a few values, because there are many rows with those values. Some buckets cover many values, because there are few rows with those values.

Height-balanced histograms are more suitable for computing selectivity estimates and are used by the Oracle9i Server.
Histograms and Selectivity

- Popular and nonpopular values
- Range scan selectivity is determined by the number of buckets spanned
- Equality search selectivity is determined by:
  - The number of buckets (popular values)
  - DENSITY (nonpopular values)

Popular and Nonpopular Values
When you create histograms, the Oracle9i Server distinguishes between popular and nonpopular values. Popular values appear more than once as a bucket endpoint value; values that appear only once or not at all are considered nonpopular.

DENSITY Statistic
When you create a histogram, the DENSITY statistic is calculated. This is calculated by 1 over NUM_DISTINCT. It is used by the CBO to reduce the bias caused by the repetition of popular values. When a column has no histograms, this statistic contains a NULL value.

CBO Selectivity Estimates
When the CBO estimates selectivity for a range scan, the number of buckets spanned is divided by the total number of buckets. The CBO does not consider partial buckets, assuming that the number of buckets offers enough granularity. For equality search conditions, the Oracle9i optimizer has two approaches:
- For a popular value, the number of buckets having that endpoint value is divided by the total number of buckets.
- For a nonpopular value, the DENSITY statistic is used as a selectivity estimate.
Example of Histogram Statistics Collection

The column statistics contain the bucket endpoints for the histogram, the number of distinct values and null values, and the density of those values.

Collection of Statistics

Use the ANALYZE command to collect information for histograms by using the FOR clause (see also the beginning of this lesson). The SIZE option specifies the number of buckets.

```
ANALYZE TABLE table {COMPUTE|ESTIMATE} STATISTICS
FOR {TABLE | ALL INDEXES
ALL [INDEXED] COLUMNS [SIZE n]
| FOR COLUMNS [SIZE n] column [SIZE n]
[,] column [SIZE n]]...}
```

Note: The maximum number of histogram buckets is 255. Note, however, that Oracle will never create more buckets than the number of distinct column values because that is useless. If you have one bucket for each distinct value, you can store the exact cardinality of each column value.

Instructor Note

Demonstration script: demo07_08.sql

This demonstration is not very useful yet. However, you can present the histogram data at the end of this lesson with a query on the data dictionary.

Oracle9i: SQL Tuning Workshop 7-22
Histogram Statistics: Example

- Generate statistics for the `PRODUCTS` table and the `PROD_LIST_PRICE` column; maximum 50 buckets using `DBMS_STATS.GATHER_TABLE_STATS`.

```sql
SQL> EXECUTE DBMS_STATS.GATHER_TABLE_STATS
2 ("sh","PRODUCTS", METHOD_OPT =>
3 'FOR COLUMNS SIZE 50 prod_list_price');
```

- The `SIZE` keyword declares the maximum number of buckets for the histogram.
- Use the `DBMS_STATS` package to have the database automatically decide which columns need histograms by specifying the `SIZE` as “auto.”

Collection of Statistics (continued)

You can generate histograms by using the `DBMS_STATS` package. You can generate histograms for columns of a table or partition. The `SIZE` keyword declares the maximum number of buckets for the histogram.

It is recommended that you use the `DBMS_STATS` package to have the database automatically decide which columns need histograms. This is done by specifying size as “auto.”
Histogram Tips

- Use the FOR ALL INDEXED COLUMNS option.
- Update histograms frequently if data distribution is not static.
- WHERE clauses with bind variables do not use histograms.
- Do not use histograms unless they substantially improve performance.
- Histograms allocate additional storage.

For many tables, it is appropriate to use the FOR ALL INDEXED COLUMNS clause to collect statistics on all indexed columns. Be aware, however, that histograms are also created on unique and primary key constrained columns that are enforced through indexes, which might be a waste of space.

If the data distribution is not static, the histogram must be frequently updated.

Because the optimizer does not take into consideration the current value of bind variables, histograms are not helpful (and therefore not worth the expense) when bind variables are used.

Histograms allocate additional storage and should be used only when they can substantially improve the query plans.
When to Use Histograms

- Use histograms for a column when the data is highly skewed.
- Avoid using histograms when:
  - The column is not used in `WHERE` clauses
  - The column is unique and used only with equality predicates
  - All predicates on the column use bind variables
  - The column data is uniformly distributed

Instructor Note

Histograms, when combined with bind variables, can have undesirable effects
Choosing a Sample Size

- If the data is uniformly distributed, 5% of the rows will suffice.
- Choose a higher sample size if the number of distinct values is more than 10% of the number of rows.
- When using histograms, the number of sampled rows must be at least 100 times the number of buckets.

Choosing a Sample Size

If the data to be analyzed is almost uniformly distributed, a sample of about 5% of the rows should be adequate. Do not use histograms in this case, because the benefit will be minimal.

If the number of distinct values is large—such as more than 10% of the number of rows—then a higher sampling percentage may be necessary.

If you decide to create histograms, the number of rows sampled should be at least 100 times the number of buckets in the histogram. In other words, the average number of sampled rows per bucket should be at least 100. You should not combine a very detailed histogram with a minimal sample size.
Choosing the Number of Buckets

- Start with the default size of 75 buckets.
- Experiment with different sizes to obtain optimum results.
- Set the number of buckets to be greater than the number of frequently occurring distinct values, if there are relatively few.

Choosing the Number of Buckets for a Histogram

The default number of buckets is 75. This value provides an appropriate level of detail for most data distributions. However, because the number of buckets in the histogram, the sample size, and the data distribution all affect the usefulness of a histogram, you may need to experiment with different numbers of buckets to obtain the best results.

If the number of frequently occurring distinct values in a column is relatively small, then you should set the number of buckets to be greater than the number of frequently occurring distinct values.

Note that the Oracle server will never create more buckets than the number of distinct column values, and the maximum is 255 buckets. It is recommended that DBMS_STATS package be used to automatically size the histograms by specifying “auto.”

You can use SQL*Plus AUTOTRACE or EXPLAIN PLAN to see the influence of the number of buckets on the cost, as estimated by the cost-based optimizer. At a certain moment, raising the number of buckets will no longer significantly change the cost calculations.

Instructor Note

Be aware of the following: Histograms on NUMBER and DATE columns are always accurate. However, histograms on alphanumeric columns like VARCHAR2 truncate the values, which causes collisions if the leading part of the strings are equal. In earlier releases only the first six characters are considered. This was relieved in release 8.1.6 and later where the first 31 characters are used.
Viewing Histogram Statistics

- **Histogram information:**

```
SQL> select endpoint_number, endpoint_value
2  from dba_tab_histograms
3  where table_name = 'PRODUCTS'
4  and column_name = 'PROD_LIST_PRICE';
```

- **The number of buckets in each histogram:**

```
USER/ALL_TAB_COL_STATISTICS
```

---

**Viewing Histogram Statistics**

The example query in the slide probably shows the following result:

```
ENDPOINT_NUMBER   ENDPOINT_VALUE
-----------------   --------------
         1           7.5
         2           8.99
         3           9.95
...
        49          198
        50         1200
          0           3
```

You have 67 buckets, including a special zero bucket to store the lowest `PROD_LIST_PRICE` values. The last bucket (excluding the 0 bucket) has 1200 as the endpoint value, which is the maximum `PROD_LIST_PRICE` value.

**Note:** If a histogram has popular values, they show up as “gaps” in the endpoint number column. Multiple buckets with the same endpoint value are stored in the data dictionary as a single row, to preserve space.
Summary

In this lesson, you should have learned how to:

- Collect table, column, index, and cluster statistics with the ANALYZE command
- Use the DBMS_STATS package
- Copy statistics between databases
- Determine selectivity for predicates with and without bind variables
- Utilize histograms
  - Popular
  - Nonpopular values

Summary

This lesson introduced you to table statistics used with the cost-based optimizer. By using the ANALYZE command or DBMS_STATS package, you can collect statistics on your tables which are used by the cost-based optimizer to determine optimal execution routes for SQL statements on analyzed tables.

Selectivity, the expected percentage of rows returned, depends on the type of operation performed in the WHERE clause. The cost-based approach is more likely to choose an index access path for a query with good selectivity. Selectivity is influenced by data distribution. When bind variables are used in a SQL statement, the optimizer must use a built-in default selectivity. These built-in values cannot be influenced.

Oracle can use a histogram to decide whether or not to use the index. A histogram stores information about the frequency of various column values. Without histograms, an even distribution is assumed. Create histograms for columns with skewed data.

Instructor Note (for page 7-28)

Demonstration script: demo07_09.sql. This shows the result of the previous demonstration when you created histograms on the PRODUCTS table. You can also use the histdemo.sql script, based on the example histogram in this lesson. That script does not show endpoint numbers 0 and 1 (see the note above).
Practice Overview

This practice covers the following topics:

- Generating table statistics
- Creating a histogram
- Calculating selectivity
- Querying data dictionary information
- Deleting table statistics

Practice Overview

In this practice, you generate table statistics for the PRODUCTS table. Because the PROD_STATUS column of the PRODUCTS table is a good candidate for a histogram, you create a histogram for that column. You calculate the selectivity of a certain predicate on the column, both before and after creating the histogram. Then you query the data dictionary for the last analyze date and sample size. Lastly, you delete the statistics on the PRODUCTS table from the data dictionary.
Practice 7

1. Turn SQL*Plus AUTOTRACE off.
   Compute statistics for the PRODUCTS table and view the table and column statistics with appropriate select statements.

2. Why is the PROD_STATUS column of the PRODUCTS table a good candidate for creating a histogram?
   **Hint:** Retrieve the (distinct) PROD_STATUS values and their cardinality.

3. Create a histogram for the PROD_STATUS column with 20 buckets and view the histogram statistics.

4. Calculate the selectivity of the following select statement, both before and after creating the histogram:
   SQL> select * from products
   2   where prod_status = 'obsolete';

5. Identify the last analyze date and sample size for all tables in your schema.

6. Delete the statistics for the PRODUCTS table and check the data dictionary again.
Influencing the Optimizer

Schedule: Timing Topic
45 minutes Lecture
10 minutes Practice
55 minutes Total

Instructor Note
This lesson starts with a recapitulation of influencing the optimizer mode at the instance and the session level. Then hints are introduced: basic access path hints, advanced access path hints, and additional hints. Finally, hints and views are covered, introducing the concept of mergeable and nonmergeable views.
Objectives

After completing this lesson, you should be able to do the following:

• Influence the optimizer behavior at the following levels:
  – Instance and session level
  – Statement level

• Specify access-path hints

• Specify hints on and in views

Note

• Influencing the optimizer at the instance and session level is also covered in the “Rule-Based Optimization Versus Cost-Based Optimization” lesson.

• When statistics are not available and cost-based optimization is forced, the optimizer uses default built-in values instead. This may produce suboptimal execution plans.
Setting the Optimizer Mode

At the instance level, set the following parameter:

```
OPTIMIZER_MODE = {CHOOSE | RULE | FIRST_ROW_n | ALL_ROWS}
```

For a session, use the following SQL command:

```
SQL> alter session set optimizer_mode = {choose | rule | first_rows_n | all_rows}
```

Setting the Optimizer Mode

Oracle9i has five basic optimizer modes:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHOOSE</td>
<td>Cost-based when statistics are available (ALL_ROWS), and rule-based when no statistics are available</td>
</tr>
<tr>
<td>RULE</td>
<td>Forces rule-based optimization regardless the presence of statistics</td>
</tr>
<tr>
<td>FIRST_ROWS_N</td>
<td>The optimizer uses a cost-based approach, regardless of the presence of statistics, and optimizes with a goal of best response time to return first $n$ number of rows. $N$ can equal 1, 10, 100, or 1000.</td>
</tr>
<tr>
<td>FIRST_ROWS</td>
<td>Optimizes response time until first result comes back (such as for interactive OLTP applications)</td>
</tr>
<tr>
<td>ALL_ROWS</td>
<td>Optimizes overall throughput (such as for reports and batch jobs)</td>
</tr>
</tbody>
</table>

Optimizer Approach Levels

At the instance level, you can set the optimizer approach by using the `OPTIMIZER_MODE` initialization parameter. When this parameter is not set, it defaults to `CHOOSE`.

At the session level, the optimizer approach set at instance level can be overruled using the `ALTER SESSION` command.

This lesson explains how to overrule the optimizer approach at the statement level by using hints.
Some Additional Parameters

- **OPTIMIZER_FEATURES_ENABLE**
  - Set to current release to enable the latest optimizer features
  - Defaults to current release of database

- **OPTIMIZER_INDEX_COST_ADJ**

Some Additional Parameters

You can use **OPTIMIZER_FEATURES_ENABLE** to change a set of initialization parameters which control the optimizer’s behavior. Values are, for example, 8.0.5, 8.0.5, 8.0.6, 8.1., 8.1.6, 8.1.7, and 9.0.1. You can use this parameter to prevent unexpected optimizer behavior when migrating to a higher version.

Setting the parameter to 9.0.1 enables these new tuning features:
- Peeking of user-defined bind variables
- Complex view merging
- Push-join predicate
- Consideration of bitmap access paths for tables with only B*-tree indexes
- Subquery unnesting
- Index joins

**Note:** If you upgrade to a new release and you want to enable the features with that release, then you do not need to explicitly set the **OPTIMIZER_FEATURES_ENABLE** parameter.
Some Additional Parameters (continued)

You can use `OPTIMIZER_INDEX_COST_ADJ` to tune the optimizer behavior for access path selection to be index-friendly. This parameter enables you to adjust the costing of index access paths in the cost-based optimizer and thereby make the optimizer prone to selecting an index access path over a full table scan.

The default for this parameter is 100 percent, which makes the optimizer estimate index access paths at the regular cost. Any other value will make the optimizer estimate the access path at that percentage of the regular cost. For example, setting it to 50 percent will make the index access path seem half as expensive as normal.

The legal range of values for this parameter is 1 to 10,000 percent. You can use this parameter to tune the performance of a system where it is felt that the optimizer chooses too few or too many index access paths.

**Note:** There are more parameters to influence the optimizer that are specific to joins; those parameters will be covered in the “Sorting and Joining” lesson.
Optimizer Hint Syntax

Enclose hints within the comments of a SQL statement. You can use either style of comment. The hint delimiter (+) must come immediately after the comment delimiter. If you separate them by a space, the optimizer will not recognize that the comment contains hints.

For example, the following hints force the tables in the FROM clause to be joined left to right:

```sql
select /*+ORDERED */ ... 
select /*+ ORDERED */ ...
```

The following attempt is treated as a normal comment because of the intervening space:

```sql
select /* +ORDERED */ ... 
```

The Oracle9i Server ignores incorrectly specified hints. However, be aware of the following:

- You never get an error message.
- Other (correctly) specified hints within the same comment are considered.
- Combinations of conflicting hints are also ignored by the Oracle9i Server.
Rules for Hints

• Place hints immediately after the first SQL keyword of a statement block.
• Each statement block can have only one hint comment, but it can contain multiple hints.
• Hints apply only to the statement block in which they appear.
• If a statement uses aliases, hints must reference aliases rather than table names.

Rules for Hints

- You must place the hint comment immediately after the first keyword (SELECT, INSERT, DELETE, or UPDATE) of a SQL statement block.
- A statement block can have only one comment containing hints, but it can contain many hints inside that comment.
- Hints apply only to the statement block in which they appear and override instance- or session-level parameters.
- A statement block is:
  - A simple SELECT, INSERT, UPDATE, or DELETE statement
  - A parent statement or a subquery of a complex statement
  - A part of a compound query using set operators (UNION, MINUS, INTERSECT)
- If a SQL statement uses aliases, then hints must reference the alias rather than the table name.

Instructor Note

Optimizer goals set at the session level with the ALTER SESSION command apply only to statements submitted directly. They seem not to affect SQL that is run from within PL/SQL. You should use hints to influence the cost-based optimizer for any SQL statements submitted from within PL/SQL. You do not, however, need to raise this issue in class.

Oracle9i: SQL Tuning Workshop 8-7
Hint Recommendations

- Use hints carefully because they imply a high maintenance load.
- Be aware of the performance impact of hard-coded hints when they become less valid.

Hint Recommendations

Use hints as a last remedy when tuning SQL statements. Be aware that the optimizer cannot change execution plans, even when better alternatives become available.

Hints may become less valid (or even invalid) when the database structure or contents change.

Note: Beware that all hints, except the RULE hint, force cost-based optimization regardless of your optimizer mode setting at the session or instance level. Hence statistics may be required to ensure an optimal execution plan.
Optimizer Hint Example

```
SQL> update ----+ INDEX(p PROD_CATAGORY_IDX)
2        products p
3  set   p.prod_min_price =
4        (select
5         (pr.prod_list_price*.95)
6         from products pr
7         where p.prod_id = pr.prod_id)
8  where p.prod_category = 'Men'
9  and   p.prod_status = 'available, on stock'
10  /
```

Optimizer Hint Syntax Example
The slide shows an example with a hint that advises the cost-based optimizer to use the index. The execution plan will look like this:

<table>
<thead>
<tr>
<th>Execution Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  UPDATE STATEMENT Optimizer=CHOOSE</td>
</tr>
<tr>
<td>(Cost=2 Card=3 Bytes=195)</td>
</tr>
<tr>
<td>1  UPDATE OF 'PRODUCTS'</td>
</tr>
<tr>
<td>2  TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'</td>
</tr>
<tr>
<td>(Cost=2 Card =3 Bytes=195)</td>
</tr>
<tr>
<td>3  INDEX (RANGE SCAN) OF 'PROD_CATEGORY_IDX'</td>
</tr>
<tr>
<td>(NON-UNIQUE) Cost=1 Card=3)</td>
</tr>
<tr>
<td>4  TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS'</td>
</tr>
<tr>
<td>(Cost=2 Card=1 Bytes=26)</td>
</tr>
<tr>
<td>5  INDEX (UNIQUE SCAN) OF 'PRODUCTS_PK' (UNIQUE)</td>
</tr>
<tr>
<td>(Cost=1 Card=291)</td>
</tr>
</tbody>
</table>

Instructor Note
Demonstration script: demo08_01.sql
Hint Categories

- **Hints for optimization mode**
  - RULE, CHOOSE
  - ALL_ROWS, FIRST_ROWS (n)
- **Hints for access path methods**
- **Hints for parallel execution**
- **Hints for join orders and operations**

Parameters to change the optimizer mode at the instance or session level (RULE, CHOOSE, ALL_ROWS, FIRST_ROWS (n)) were discussed in the “Rule-Based Optimization Versus Cost-Based Optimization” lesson. You can specify these four values as a hint at the statement level as well.

**Hints for Access Path Methods**
This lesson concentrates on this hint category.

**Hints for Parallel Execution**
This topic is covered in a separate course.

**Hints for Join Orders and Operations**
Optimizing joins is covered in the “Sorting and Join Techniques” lesson.
## Basic Access Path Hints

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL(t)</td>
<td>Chooses a full table scan</td>
</tr>
<tr>
<td>ROWID(t)</td>
<td>Chooses a table scan by ROWID</td>
</tr>
<tr>
<td>INDEX(t [idx]...)</td>
<td>Chooses an (ascending) index scan</td>
</tr>
<tr>
<td>INDEX_ASC(t [idx]...)</td>
<td>Chooses an ascending index scan</td>
</tr>
<tr>
<td>INDEX_DESC(t [idx]...)</td>
<td>Chooses a descending index scan (This hint has no effect when accessing multiple tables; such statements always perform ascending range scans.)</td>
</tr>
<tr>
<td>AND_EQUAL</td>
<td>Chooses an execution plan that uses an access path that merges the t id [idx]...scans on several single-column indexes (minimum 2, maximum 5)</td>
</tr>
<tr>
<td>INDEX_FFS(t [idx]...)</td>
<td>Causes a fast full index scan to be performed</td>
</tr>
<tr>
<td>NO_INDEX(t [idx]...)</td>
<td>Explicitly disallows a set of indexes for the specified table</td>
</tr>
</tbody>
</table>

**Note:** The NO_INDEX hint is useful if you use distributed query optimization. It applies to function-based, B*-tree, bitmap, cluster, and domain indexes. If this hint does not specify an index name, the optimizer will not consider a scan on any index on the table.
Specifying Index Names

An index hint can optionally specify one or more index names:

- If this hint specifies a single available index, the optimizer performs a scan on this index. The optimizer does not consider a full table scan or a scan on another index on the table.
- If this hint specifies a list of available indexes, the optimizer considers the cost of a scan on each index in the list and then performs the index scan with the lowest cost. The optimizer may also choose to scan multiple indexes from this list and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan or a scan on an index that is not listed in the hint.
- If this hint specifies no indexes, the optimizer considers the cost of a scan on each available index on the table and then performs the index scan with the lowest cost. The optimizer might also choose to scan multiple indexes and merge the results, if such an access path has the lowest cost. The optimizer does not consider a full table scan.

Fast Full Index Scans

Fast full index scans are an alternative to full table scans when the index contains all the columns that are needed for the query. It cannot be used to eliminate a sort operation. It reads the entire index by using multiblock reads (as in a full table scan).

Note: During a full table scan Oracle can read multiple blocks into memory in a single I/O operation. This is commonly referred to as multiblock reads. It is controlled by the DB_FILE_MULTIBLOCK_READ_COUNT initialization parameter. This parameters values is OS dependent.

Instructor Note (for the next page)

Demonstration script demo08_02.sql.

Purpose: This script first drops all indexes on the CUSTOMERS table, then creates an index on the COUNTRY_ID column. Several SELECT statements are run, some including hints. After each SELECT statement, the screen is paused for you to explain the results before moving to the next statement.
## Advanced Access Path Hints

<table>
<thead>
<tr>
<th>Hint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLUSTER(t)</td>
<td>Performs a cluster scan (applies to index clusters only)</td>
</tr>
<tr>
<td>HASH(t)</td>
<td>Performs a hash scan (applies to hash clusters only)</td>
</tr>
<tr>
<td>HASH_AJ(t)</td>
<td>Transforms a <code>NOT IN</code> subquery into a hash anti-join</td>
</tr>
<tr>
<td>MERGE_AJ(t)</td>
<td>Transforms a <code>NOT IN</code> subquery into a merge anti-join</td>
</tr>
<tr>
<td>MERGE_SJ(t)</td>
<td>Transforms a correlated <code>EXISTS</code> subquery into a merge semijoin</td>
</tr>
<tr>
<td>USE_CONCAT</td>
<td>Rewrites <code>OR</code> into <code>UNION ALL</code> and turns off inlist processing and <code>OR</code>-expands all disjunctions</td>
</tr>
<tr>
<td>NO_EXPAND</td>
<td>Prevents <code>OR</code>-expansions (This hint is the opposite of USE_CONCAT.)</td>
</tr>
</tbody>
</table>

**Anti-Joins and Semijoins**

These two techniques to change subquery statements into equivalent joins are explained in the “Sorting and Joining” lesson.

*Oracle9i: SQL Tuning Workshop 8-13*
Buffer Cache Hints

The **CACHE** hint specifies that the blocks retrieved for this table are placed at the most recently used (MRU) end of the LRU list in the buffer cache when a full table scan is performed. This option is useful for small lookup tables.

The **NOCACHE** hint specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the normal behavior of the buffer cache, ensuring that a single full table scan of a large table cannot push all recently used blocks from the LRU list.

**Note:** You can also specify the **CACHE** or **NOCACHE** attribute when creating or altering tables.

**Instructor Note**

Until release 8.0, you could use the `CACHE_SIZE_THRESHOLD` parameter to determine what is considered a small table in this context. This parameter was eliminated in Oracle8i, featuring multiple buffer pools. Check the “table scans (long tables)” and “table scans (short tables)” statistics in `V$SYSSTAT` to see the effects of the **CACHE** and **NOCACHE** hints.

The **CACHE** and **NOCACHE** hints are being deprecated in favor of using multiple buffer pools.
Hints and Views

- Do not use hints in views.
- Use view optimization techniques:
  - Statement transformation
  - Access view results as a table
- Hints can be used on mergeable views and nonmergeable views.

Hints and Views

You should not use hints inside or on views. This is because views can be defined in one context and used in another; such hints can result in unexpected plans. In particular, hints inside views or on views are handled differently depending on whether or not the view is mergeable into the top-level query.

**View Optimization**

To optimize a statement that accesses a view, the optimizer has two alternatives. Normally the statement is transformed into an equivalent statement that accesses the view’s base tables. The optimizer can use one of these techniques to transform the statement:

- Merge the view’s query into the referencing query block in the accessing statement
- Push the predicate of the referencing query block inside the view

When these transformations are impossible, the view’s query is executed and the result is accessed as though it were a table. This appears as a **VIEW** step in execution plans.
Hints and Views (continued)

Mergeable Views

The optimizer can merge a view into a referencing query block, provided the view definition does not contain the following:

- Set operators (UNION, UNION ALL, INTERSECT, MINUS)
- A CONNECT BY clause
- The ROWNUM pseudocolumn
- Group functions (AVG, COUNT, MAX, MIN, SUM) in the select list

Hints and Mergeable Views

Optimization approach and goal hints can occur in a top-level query or inside views:

- If there is such a hint in the top-level query, that hint is used regardless of any other such hints inside the views.
- If there is no top-level optimizer-mode hint, then mode hints in referenced views are used as long as all mode hints in the views are consistent.
- If two or more mode hints in the referenced views conflict, then all mode hints in the views are discarded and the session mode is used, whether default or user-specified.

Access method and join hints on referenced views are ignored unless the view contains a single table (or references another view with a single table). For such single-table views, an access-method hint or a join hint on the view applies to the table inside the view.

Access method and join hints can also appear in a view definition:

- If the view is a subquery (that is, if it appears in the FROM clause of a SELECT statement), then all access-method and join hints inside the view are preserved when the view is merged with the top-level query.
- For views that are not subqueries, access-method and join hints in the view are preserved only if the top-level query references no other tables or views (that is, if the FROM clause of the SELECT statement contains only the view).

Hints and Nonmergeable Views

With nonmergeable views, optimization mode hints inside the view are ignored. The top-level query decides the optimization mode.

Because nonmergeable views are optimized separately from the top-level query, access-method and join hints inside the view are always preserved. For the same reason, access-method hints on the view in the top-level query are ignored.

However, join hints on the view in the top-level query are preserved because in this case, a nonmergeable view is similar to a table.
View Processing Hints

<table>
<thead>
<tr>
<th>MERGE</th>
<th>Merges complex views or subqueries with the surrounding query</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_MERGE</td>
<td>Does not merge mergeable views</td>
</tr>
</tbody>
</table>

You can cause a view to be merged on a per-query basis by using the MERGE and NO_MERGE hints. The NO_MERGE hint causes the Oracle9i Server not to merge mergeable views.

**Example**

```sql
SQL> select /*+ MERGE(v) */
2       p.prod_id, p.prod_name, v.sold
3  from   products p,
4       (select s.prod_id, sum(amount_sold) as sold
5          from sales s
6          group by s.prod_id)  v
7  where  p.prod_id = v.prod_id
8  and p.prod_status = 'obsolete';
```

**Note:** Because the group function is used on line 6, the MERGE (v) hint is ignored.

**Instructor Note**

Demonstration script: `demo08_03.sql` to show that the first code example cannot be merged. Starting with version 8.1.6, global hints are available. You can use them to specify hints at the upper query level that are meant for views accessed by that query, so that you do not need to specify hints in the view definition. This is done with a special dot notation syntax; see *Oracle9i Database Performance Guide and Reference* for details.

The SALES table is a partitioned table. For an explanation of partitioning, refer students to the documentation.
Summary

In this lesson, you should have learned how to:

- Set the optimizer mode
- Use optimizer hint syntax
- Determine access path hints
- Analyze hints and their impact on views

Summary

This lesson introduced you to additional optimizer settings and hints.

You can set the optimizer mode to use either the cost-based method or rule-based method. The supported optimizer mode values are `CHOOSE`, `RULE`, `FIRST_ROWS(n)`, and `ALL_ROWS`.

By using hints, you can influence the CBO at the statement level. Use hints as a last remedy when tuning SQL statements. There are several hint categories, one of which is hints for access path methods.

To specify a hint, use the hint syntax within the SQL statement. Alternatively, you can use the Oracle SQL Analyze graphical interface to specify a hint within SQL statements.

Be careful when using hints with views. Hints inside views or on views are handled differently depending on whether or not the view is mergeable into the top-level query.

Instructor Note

After this lesson, students can complete Workshop 2 (sorting, grouping, and set functions). Note that some theory about sorting is discussed in the beginning of the next lesson.
Practice Overview

This practice covers the following topics:

• Analyzing a query on a table without any statistics
• Using a hint to force a full table scan

Practice Overview

In this practice, you investigate RBO behavior for a query on the PROMOTIONS table, which is a small table with an index. This is accomplished by deleting table statistics and setting the optimizer mode to CHOOSE. Then you use a hint to force a full table scan on the table. Compare the results and decide the best approach for this query.
Practice 8

1. Delete all statistics for the PROMOTIONS table.

2. Display the contents of the PROMOTIONS table. Note that it is a relatively small table. There is an index on the PROMO_ID column, to support the primary key constraint.

3. Alter your session and set the optimizer mode to CHOOSE.

4. Analyze the select statement shown.

   SQL> select promo_id, promo_name, promo_cost
      2 from promotions
      3 where promo_id > 450;

5. What optimization method is used?

6. Include a hint to use a full table scan with the select statement used in step 4.

7. View the results of the EXPLAIN PLAN command. What optimization method is used? With the PROMOTIONS table, which approach is more efficient?
Sorting and Joining

Schedule: Timing Topic
         90 minutes Lecture
         90 minutes Total

Instructor Note
This lesson is long, and includes a lot of detail about join techniques. Probably the best approach is to teach the basics (until “Join Order Rules”), and have the participants perform Workshop 3 (“Joins”). Then continue this lesson, and have them continue with Workshop 4 (“Subqueries”) at the end of the lesson.
Objectives

After completing this lesson, you should be able to do the following:

- Optimize sort performance
- Describe different join techniques
- Explain join optimization
- Find optimal join execution plans

Joins are probably the most frequent cause of performance problems.

Instructor Note

For more details about hash joins, see Oracle9i Performance and Tuning Guide.

Instructor Note (for page 9-4)

Demonstration script: demo09_01.sql

Instructor Note (for page 9-5)

Demonstration script: Run the demo09_02.sql script to demonstrate the code shown.
Tuning Sort Performance

- Sorting large sets can be expensive:
  Tune sort parameters.
- Note that `DISTINCT`, `GROUP BY`, and most set operators cause implicit sorts.
- Minimize sorting by one of the following:
  - Try to avoid `DISTINCT` and `GROUP BY`
  - Use `UNION ALL` instead of `UNION`
  - Enable index access to avoid sorting

You should contact your database administrator to tune sort parameters such as the following:

- `SORT_MULTIBLOCK_READ_COUNT`
- `SORT_AREA_SIZE`
- `SORT_AREA_RETAINED_SIZE`

For more details about these parameters, check the *Oracle9i Reference*.

If possible, write SQL statements in such a way that sorting is not needed. Note that several SQL language components cause implicit sorts, such as `DISTINCT`, `GROUP BY`, `UNION`, `MINUS`, and `INTERSECT`. Make sure that you get only the sorts that are needed for your result.

Sorts can be avoided by creating indexes. Concatenated indexes, with the appropriate `DESC` or `ASC` attributes, can especially help avoid sorts. Note, however, that you lose the advantage of multiblock I/O, because to avoid a sort the index must be scanned sequentially. An index fast full scan (using multiblock I/O) does not guarantee that the rows are returned in the right order.

**Instructor Note**

The students have already investigated some sort behavior in Workshop 2 (sorting, grouping, and set functions), such as implicit sorts and index access to avoid sorting.
Top-N SQL

The idea behind the Top-N SQL feature is that you do not need to sort a full set if, for example, you are only interested in the four highest (or lowest) values.

If a set is too big to be sorted in memory, the performance is significantly and especially degrading. This is caused by the I/O to and from temporary storage of intermediate results on disk.

If you only want to know the four highest values, you only need an array with four slots to scan the set and keep the four highest values in that array.

The WHERE clause of the statement above is merged into the in-line view to prevent the full PRODUCTS table from being sorted by PROD_LIST_PRICE. This appears in the execution plan as follows:

Execution Plan

0      SELECT STATEMENT Optimizer=CHOOSE (Cost=248 Card=5 Bytes=660000)
1  0      COUNT (STOPKEY)
2  1      VIEW (Cost=248 Card=10000 Bytes=660000)
3  2      SORT (ORDER BY STOPKEY) (Cost=248 Card=10000 Bytes=370000)
4  3      TABLE ACCESS (FULL) OF ‘PRODUCTS’ (Cost=109 Card=10000 Bytes=370000)

Oracle9i: SQL Tuning Workshop 9-4
Join Terminology

- Join statement
- Join predicate, nonjoin predicate
- Single-row predicate

A join statement is a select statement with more than one table in the FROM clause.

However, the optimizer uses the techniques usually associated with joins when executing other kinds of statements where more than one table is referenced, such as in subqueries. Subqueries are discussed later in this lesson. This is an example of a SQL:1999 compliant join statement which is available in Oracle9i:

```
SQL> SELECT c.cust_last_name, c.cust_first_name,
          2      co.country_id, co.country_name
          3  FROM   customers c JOIN countries co
          4  ON     (c.country_id = co.country_id) -- join predicate
          5  AND    co.country_id = 'JP'        -- nonjoin predicate
          6  OR     c.cust_id = 205;         -- single-row predicate
```

A join predicate is a predicate in the WHERE clause that combines the columns of two of the tables in the join.

A nonjoin predicate is a predicate in the WHERE clause that references only one table.

A single-row predicate is an equality predicate on a column with a unique or primary key constraint, or a column with a unique index without corresponding constraint. The optimizer knows that these predicates always return one row or no rows at all.
Join Terminology

• Equijoin

```
SQL> SELECT c.cust_last_name, co.country_name
2  FROM customers c, countries co
3  WHERE c.country_id = co.country_id;
```

• Nonequijoin

```
SQL> SELECT c.cust_last_name, c.cust_credit_limit, cr.credit_limit_id
2  ,       cr.credit_limit_id
3  FROM customers c, credit_limits cr
4  WHERE c.cust_credit_limit
5  BETWEEN cr.credit_limit_min
6  AND     cr.credit_limit_max;
```

Join Terminology (continued)

An *equijoin* is a join statement where the join predicate shows the equal (=) sign between the columns. An equijoin is the most common type of join. The statement above is an example of an equijoin.

A *nonequijoin* refers to join statements where the join predicate uses one of the other comparison operators, such as <, >, <=, >=, and <>. Nonequijoins are also called theta-joins. They occur infrequently and may indicate a poor database design.

Instructor Note

The CREDIT_LIMITS table is not part of the standard SH schema. Run the demo09_03.sql script to create the CREDIT_LIMITS table, insert rows, and run the nonequijoin statement shown above.

Note the full table scan on both tables.

In Workshop 3, the students investigate an example of a nonequijoin.
Join Operations

A join operation combines the output from two row sources and returns one resulting row source. Row sources were introduced in the “EXPLAIN and AUTOTRACE” lesson.

Join Operation Types

The Oracle9i Server supports different join operation types:

- Nested loops join
- Sort/merge join
- Hash join
- Cluster join
- Full outer Joins

Instructor Note

Oracle9i also supports index joins as a new access path. The INDEX_JOIN hint instructs the optimizer to use that path. This hint has an effect if a sufficiently small number of indexes exist that (together) contain all columns required to resolve the query, so that access to that table can be avoided. So it is not a different join technique, but rather a smart way to avoid table access by using multiple indexes.
Nested Loops Joins

- One of the two tables is defined as the *outer* table or the *driving* table.
- The other table is called the *inner* table.
- For each row from the outer table, all matching rows in the inner table are retrieved.

<table>
<thead>
<tr>
<th>For each row in the outer table</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each row in the inner table</td>
</tr>
<tr>
<td>Check for a match</td>
</tr>
</tbody>
</table>

Basic Execution Plan of Nested Loops Joins

```sql
NESTED LOOPS
  TABLE ACCESS (...) OF our_outer_table
  TABLE ACCESS (...) OF our_inner_table

Because they are often used with indexed table accesses, nested loops are common with an execution plan that looks like following:

NESTED LOOPS
  TABLE ACCESS (BY ROWID) OF our_outer_table
  INDEX (... SCAN) OF outer_table_index (.....)
  TABLE ACCESS (BY ROWID) OF our_inner_table
  INDEX (RANGE SCAN) OF inner_table_index (NON-UNIQUE)
```

It is common that the outer (driving) table is accessed with a full table scan.

Nonequijoins

If you have a nonequijoin, a hash join is not possible. Depending on the nonjoin predicates, nonequijoins can result in a nested loop with full table scans on both the outer and the inner tables. This means that a full table scan is performed on the outer table; and for each row, a full table scan is performed (again and again) on the inner table. This usually results in poor performance.
Nested Loops Join Plan

For each row from the outer (driving) table, all rows from the inner table that satisfy the join predicate are retrieved.

Any nonjoin predicates on the inner table are considered after this initial retrieval, unless a composite index, which combines both the join and the nonjoin predicate, is used.

The nested loops join can solve all kinds of joining requirements, no matter what the join predicate is. Therefore, you should use the nested loops join when you cannot use sort/merge, hash, or cluster joins. These join techniques are discussed later in this lesson.
Sort/Merge Joins

A sort/merge join is executed as follows:

- The rows from each row source are sorted on the join predicate columns
- The two sorted row sources are then merged and returned as the resulting row source

Sort/Merge Joins

In the sort operations, the two row sources are sorted on the values of the columns used in the join predicate. If a row source has already been sorted in a previous operation, the sort merge operation skips the sort on that row source.

Sorting could make this join technique expensive, especially if sorting cannot be performed in memory.

The merge operation combines the two sorted row sources to retrieve every pair of rows that contain matching values for the columns used in the join predicate.

Basic Execution Plan

MERGE (JOIN)
  SORT (JOIN)
    TABLE ACCESS (...) OF tableA
  SORT (JOIN)
    TABLE ACCESS (...) OF tableB
Sort/Merge Join Plan

The table accesses (row sources 4 and 5) can be based on index scans if there are nonjoin predicates that can use an index. However, this join operation most often appears with full table scans on both tables.

The sequence (which table is sorted first) does not matter; there is no concept of an outer/driving table. Remember that the sorting can result in the creation of temporary segments on disk (giving more I/O than just the table scans themselves).

Sort/merge joins tend to perform better than an indexed nested loops join if the number of rows satisfying the join condition represents a major part of the total number of rows in the two tables.

Note: The break-even point for a sort/merge join and an indexed nested loops join depends on the sorting performance of the database. Sort performance can be influenced by defining temporary tablespaces of tablespace type TEMPORARY, setting the SORT_AREA_SIZE parameter to define the amount of memory to use for a sort, and setting SORT_AREA_RETAINED_SIZE to release memory after a sort. This is typically a DBA task.
Hash Joins

A hash join is executed as follows:
- Both tables are split into as many partitions as required, using a full table scan.
- For each partition pair, a hash table is built in memory on the smallest partition.
- The other partition is used to probe the hash table.

The Oracle9i Server considers hash joins only when you use the cost-based optimizer. Hash joins, like sort/merge joins, can be performed only for equijoins. These are the steps performed for a hash join:
- The Oracle server performs a full table scan on both tables and splits each into as many partitions as required, based on the available memory.
- The Oracle server builds a hash table on the smallest partition and uses the other partition to probe the hash table. All partition pairs that do not fit into memory are placed onto disk.

The number of partitions into which the tables are split depends on the amount of available memory.

**Basic Execution Plan**

```
HASH JOIN
    TABLE ACCESS (..) OF tableA
    TABLE ACCESS (..) OF tableB
```

**Performance Considerations**

As a general rule, hash joins outperform sort/merge joins. In some cases, a sort/merge join can outperform a nested loops join; it is even more likely that a hash join will.
Hash Join Plan

To execute a hash join, the Oracle server performs these steps:

- Steps 2 and 3 perform full table scans of the tables that must be joined.
- Step 1 builds a hash table out of the rows coming from step 2 and probes it with each row coming from step 3.

**Note:** The `HASH_AREA_SIZE` initialization parameter controls the amount of memory used for hash join operations and the `HASH_MULTIBLOCK_IO_COUNT` initialization parameter controls the number of blocks that a hash join operation should read and write concurrently.
Joining Multiple Tables

You can join only two row sources at a time. Joins with more than two tables are executed as follows:

1. Two tables are joined, resulting in a row source.
2. The next table is joined with the row source resulting from step 1.
3. Repeat step 2 until all tables are joined.
Outer Joins

- Join predicates with a plus (+) sign
- Nonjoin predicates with a plus (+) sign
- Predicates without a plus (+) sign will disable outer join functionality. initialization parameter

```sql
SQL> SELECT s.time_id, t.time_id
  2  FROM sales s, times t
  3  WHERE s.time_id (+) = t.time_id;  
```

Note: Predicates without a plus (+) sign on the table that is outer joined will disable the outer join functionality.

Instructor Note
Demonstration script: Run the `demo09_04.sql` script to demonstrate the code shown. The first example shows the outer join disabled, the second join shows the outer join enabled.
Outer Joins

In Oracle9i SQL:1999 compliant join syntax has been introduced. So now you can specify RIGHT, LEFT, and FULL OUTER joins to create the outer joins instead of using the (+) sign. This is not intended to improve performance. The usage of (+) is still supported.

**Note:** This statement is the equivalent to the statement in the previous slide.
Full Outer Joins

- A full outer join acts like a combination of the left and right outer joins.
- In addition to the inner join, rows from both tables that have not been returned in the result of the inner join are preserved and extended with nulls.
- This syntax has been introduced in Oracle9i.

SQL> SELECT c.cust_id, c.cust_last_name, co.country_name
FROM customers c
FULL OUTER JOIN countries co
ON (c.country_id = co.country_id);

Outer Joins

A full outer join acts like a combination of the left and right outer joins. In addition to the inner join, rows from both tables that have not been returned in the result of the inner join are preserved and extended with nulls. In other words, with full outer joins you can join tables together, yet still show rows which do not have corresponding rows in joined-to tables.

For both the cost-based and rule-based approaches, the optimizer first determines whether joining two or more of the tables definitely results in a row source containing at most one row. The optimizer recognizes such situations based on UNIQUE and PRIMARY KEY constraints on the tables. If such a situation exists, then the optimizer places these tables first in the join order. The optimizer then optimizes the join of the remaining set of tables.

The CBO, generates a set of execution plans based on the possible join orders, join methods, and available access paths. The optimizer then estimates the cost of each plan and chooses the one with the lowest cost. The optimizer estimates costs in these ways:

The cost of a nested loops operation is based on the cost of reading each selected row of the outer table and each of its matching rows of the inner table into memory. The optimizer estimates these costs using the statistics in the data dictionary. The cost of a sort merge join is based largely on the cost of reading all the sources into memory and sorting them. The CBO’s choice of join orders can be overridden with the ORDERED hint. If the ORDERED hint specifies a join order that violates the rule for an outer join, then the optimizer ignores the hint and chooses the order. Also, you can override the optimizer’s choice of a join method with hints.

Oracle9i: SQL Tuning Workshop 9-17
Execution of Outer Joins

- Indexes can be used for outer join predicates.
- Nonjoin predicates on the outer joined table can use indexes only if no index is used for the join predicate.

```
SQL> SELECT c.cust_id, co.country_name
  2  FROM customers c , countries co
  3  WHERE c.country_id(+) = co.country_id;
```

The optimizer chooses to use a HASH_JOIN.

```
SQL> SELECT c.cust_id, co.country_name
  2  FROM customers c , countries co
  3  WHERE c.country_id(+) = co.country_id
  4  AND co.country_id = 'IT';
```

Instructor Note
Demonstration script: demo09_05.sql
The Optimizer and Joins

The optimizer decides about the following:
1. The order in which to join the tables
2. Which join operation to apply for each join
3. The access path for each row source

Instructor Note
Do not go into too much detail here. This is covered in more detail in the “Everything You Always wanted to Know about the Oracle Optimizer” seminar (65320GC10), by using event 10053.
Join Order Rules

Rule 1

A single-row predicate forces its row source to be placed first in the join order.

Rule 2

For outer joins, the table with the outer join operator (+) must come after the other table in the join order for processing the join.

Join Order Rules

- A single-row predicate forces its row source to be placed first in the join order.
- For outer join statements, the table with the outer join operator must come after (in the join order) the other table in the join predicate.

By making a single-row predicate table the driving table of a join operation, the optimizer effectively eliminates one join operation. For example, for a nested loops join, the main loop is not a real loop anymore. The single-row predicate results in one row (or no rows at all), so the inner loop is executed only once (or not at all).

The reason for the second rule (about outer joins) is that you would otherwise be unable to produce the correct outer join results. For example, if you join the COUNTRIES table with the CUSTOMERS table and you also want to see countries that have no corresponding customers, you must start with the COUNTRIES table.

After considering these two rules, determine the join order and join operations based on the optimizer goal that is used.
RBO Join Optimization

- As a first step, all possible join orders are generated.
- This potentially gives the following:

<table>
<thead>
<tr>
<th>Number of Tables</th>
<th>Join Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2! = 2</td>
</tr>
<tr>
<td>3</td>
<td>3! = 6</td>
</tr>
<tr>
<td>4</td>
<td>4! = 24</td>
</tr>
</tbody>
</table>

- Parse time grows exponentially when adding tables to a join.

Instructor Note

The exclamation mark used on the slide is the factorial operator from mathematics.

<table>
<thead>
<tr>
<th>Number of Tables</th>
<th>Possible Plans</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3! = 3 * 2 = 6</td>
</tr>
<tr>
<td>4</td>
<td>4! = 4 * 3 * 2 = 24</td>
</tr>
<tr>
<td>5</td>
<td>5! = 5 * 4 * 3 * 2 = 120</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Because only the execution plans that follow a join predicate are considered, the number of possible execution plans usually does not reach this value when the number of tables is greater than 3. However, joining 20 tables still results in many possible execution plans. Remember that all tables involved in a join, including those that are hidden in views, count as separate tables.

Instructor Note

The exclamation mark used on the slide is the factorial operator from mathematics.
RBO Join Optimization

The following rules are applied in order:
1. Minimize nested loops operations in which the inner table is accessed with a full table scan.
2. Tie: Minimize sort/merge operations.
3. Still a tie: Choose a driving table with the best-ranked access path.
4. Still a tie: Choose a driving table that appears later in the FROM clause.

RBO Join Optimization (continued)
These are the rules applied by the rule-based optimizer, in the specified order:
1. Try to minimize nested loops operations in which the inner table is accessed with a full table scan.
2. If there is a tie, try to minimize sort/merge operations.
3. If there is still a tie, choose a driving table with the best-ranked access path.
4. If there is still a tie, choose a driving table that appears later in the FROM clause (the right-to-left rule).
CBO Join Optimization

- A set of possible execution plans is generated, limited by two parameters:
  - OPTIMIZER_SEARCH_LIMIT
  - OPTIMIZER_MAX_PERMUTATIONS
- The optimizer then estimates the cost of each plan and chooses the one with the lowest cost.

CBO Join Optimization

The number of different execution plans considered is limited by the value of the OPTIMIZER_SEARCH_LIMIT parameter, which specifies the maximum number of tables for which the optimizer considers a Cartesian product. The default value is 5. As soon as you join more tables, the optimizer considers only regular joins; that is, it considers only join orders where the tables are connected by equijoin predicates.

The OPTIMIZER_MAX_PERMUTATIONS initialization parameter directly limits the number of permutations of the join order that the optimizer considers. The default value is 80,000. By reducing this value, you can ensure that parse times for queries joining lots of tables stay within acceptable limits; the disadvantage is that the optimizer may miss the optimal plan.
Estimating Join Costs

- **Nested loops join:**
  
  Cost » read(A) + card(A) * read(B)
  
  - OPTIMIZER_INDEX_CACHING parameter

- **Sort/merge join:**

  Cost » read(A) + sort(A) + read(B) + sort(B)

Estimating Join Costs

The cost of a nested loops join is based on the cost of reading each selected row of the outer table and each of its matching rows of the inner table.

The cost of a sort/merge join is based on the cost of reading and sorting both row sources. The cost-based optimizer also considers other factors when determining the cost of each operation. For example:

- A smaller sort area size is likely to increase the cost for a sort/merge join because sorting takes more CPU time and I/O in a smaller sort area.
- A larger multiblock read count is likely to decrease the cost for a sort/merge join in relation to a nested loops join. If a large number of sequential blocks can be read from disk in a single I/O, an index on the inner table for the nested loops join is less likely to improve performance over a full table scan.

Another important parameter that influences the CBO cost estimates for nested loops joins is OPTIMIZER_INDEX_CACHING. The range of values is 0–100, and the default value is 0. By setting this parameter to a higher value, the CBO will assume that any indexes on the inner table will be in the buffer cache. This means that a nested loops join will be considered to be more attractive.

**Instructor Note**

The curly equal sign (») is used in the slide to denote “is approximately equal to.” Only the main cost components are mentioned in the formulas.
Star Joins

One type of data warehouse design centers on what is known as a star schema, which is characterized by one or more very large fact tables that contain the primary information in the data warehouse and a number of much smaller dimension tables (or lookup tables), each of which contains information about the entries for a particular attribute in the fact table.

A star join is a join between a fact table and a number of lookup tables. Each lookup table is joined to the fact table by a primary-key to foreign-key join, but the lookup tables are not joined to each other. The fact table normally has a concatenated index on the key columns to facilitate this type of join.

A star join is executed using the normal join operations, but using a join order that does not correspond to the join predicates: the smaller dimension tables are joined together first (like a Cartesian product). The resulting row source is then used for a concatenated index access into the fact table using a indexed nested loops join.

Note: The star transformation, which is discussed in the “Advanced Indexes” lesson, should not be confused with the star joins. Whereas star joins work well for schemas with a small number of dimensions and dense fact tables, star transformation works well for schemas with a large number of dimensions and sparse fact tables.
Star Join Example

SQL> SELECT amount_sold, prod_name, promo_name,
  2  FROM   sales s, products p, promotions o
  3  WHERE  s.prod_id = p.prod_id
  4  AND    s.promo_id = o.promo_id;

In this example, you must construct a concatenated index on the columns PROD_ID and PROMO_ID of the SALES table to enable star query optimization.

Note: Star query optimization is considered only when using the cost-based optimizer (CBO). Star joins usually outperform the corresponding normal multitable join because the optimizer can start with a small dimension table as the driving table, but must join that table with the (large) fact table. This produces a large intermediate result that must be joined with all remaining dimension tables. The advantage of a star join is that the fact table is the last table in the join order.

Instructor Note
It seems that the concatenated index is no longer needed since release 8.0.5.
Hints for Join Orders

Specifying a hint forces the CBO to be used. Note that specifying a hint such as an index hint for one of the tables in the join results in cost-based optimization to be used for the other tables in the join, as well.

**ORDERED()**
The ORDERED() hint joins the tables in the order in which they appear in the FROM clause, from left to right. You can experiment with different join orders by changing the table order in the FROM clause.

**LEADING()**
The LEADING() hint causes the optimizer to use the specified table as the first table in the join order. The difference with the ORDERED hint is that LEADING determines only the driving table.

**STAR**
The STAR hint forces a star join plan to be used if possible. A star join plan has the largest table in the query last in the join order and joins it with a nested loops join on a concatenated index. The STAR hint applies when there are at least three tables, the large table’s concatenated index has at least three columns, and there are no conflicting access or join method hints. The optimizer also considers different permutations of the small tables.

**Instructor Note**
The LEADING hint was introduced in Oracle8i release 8.1.6.
Hints for Join Operations

In the slide, the three hints to influence the join operation accept one or more table names (or aliases). It is important to know that the table name (or alias) you specify should be the *inner* table, not the driving table, for the join operation you want to be performed. See also the last note on the next page.

Also note that the `USE_HASH` and `USE_MERGE` hints are always ignored for nonequijoins, because nonequijoins can be resolved only with a nested loops join operation.

**Examples**

The `USE_NL()` hint causes each specified table to be joined with another row source with a nested loops join using the specified table as the inner table:

```
SQL> select /*+ USE_NL(T2) */...
  2  from   T1, T2, ...
  3  where ...
```

```
SQL> select /*+ USE_MERGE(T2) */...
  2  from   T1, T2, ...
  3  where ...
```

```
SQL> select /*+ USE_HASH(T2) */...
  2  from   T1, T2, ...
  3  where ...
```

This should give the following execution plan structure:

```
NESTED LOOPS
  TABLE ACCESS (...) OF T1
NESTED LOOPS
  TABLE ACCESS (...) OF T2
  TABLE ACCESS (...) OF T3
```
Hints for Join Operations (Continued)

Examples (continued)

The USE_MERGE hint causes each specified table to be joined with the previous row source in the join order with a sort-merge join:

```
SQL> select /*+ USE_MERGE(TAB2) */ ... 
   2  from TAB1, TAB2 
   3  where ...
```

The USE_HASH hint causes each specified table to be joined with the previous row source in the join order with a hash join:

```
SQL> select /*+ USE_HASH(TAB2) */ ... 
   2  from TAB1, TAB2 
   3  where ...
```

Using the USE_NL(), USE_MERGE(), and USE_HASH() Hints

The Oracle9i Server uses these three join operation hints when the referenced table (or alias) is forced to be the inner table of a join, and they are ignored if the referenced table (or alias) is the outer table. That is why these hints are often combined with the ORDERED hint.

If you have the idea that the optimizer is ignoring your USE_NL, USE_MERGE, or USE_HASH hints, the Oracle9i optimizer probably found a better plan for a different join order.

To eliminate this possibility, you can add the ORDERED hint and change the order of the tables in your FROM clause, to force a certain join order.

It is possible to specify two arguments in the USE_HASH() and USE_MERGE() hints when both row sources are tables instead of a table being joined to a result set.
Other Join Hints

**STAR_TRANSFORMATION**

The hint in the slide is not really a join hint, but rather a hint to optimize data access for the facts table of a star schema. Refer to the “Advanced Indexes” lesson for a further description of this hint.

**DRIVING_SITE**

The DRIVING_SITE hint forces query execution to be done at a different site than that selected by the Oracle9i Server. This hint can be used with either rule-based or cost-based optimization.

**Example**

```
SQL> SELECT /*+ DRIVING_SITE(T1) */
2  FROM T1, T2, ...
3  WHERE ...
```

**Instructor Note**

Check whether your students are familiar with the `countries@rsite` notation for remote tables; `rsite` is a database link reference.
Subqueries and Joins

- Subqueries, like joins, are statements that reference multiple tables.
- Subquery types:
  - Noncorrelated subqueries
  - Correlated subqueries
  - \texttt{NOT IN} subqueries (anti-joins)
  - \texttt{EXISTS} subqueries (semijoins)

Noncorrelated Subqueries

A noncorrelated subquery does not contain any references to the outer (main) query and can be executed independently. For example:

\begin{verbatim}
SQL> SELECT c.*
    2  FROM   customers c
    3  WHERE  c. country_id IN
    4        (SELECT co.country_id
    5           FROM   countries co
    6           WHERE  co.country_subregion = 'Asia');
\end{verbatim}

This statement is executed as a hash join with the subquery as the outer table. It retrieves all customers in location Asia. This is the execution plan structure:

\begin{verbatim}
Execution Plan
---------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=363 Card=6250
Bytes= 918750)
1    0   HASH JOIN (Cost=363 Card=6250 Bytes=918750)
2    1     TABLE ACCESS (FULL) OF 'COUNTRIES' (Cost=1 Card=2
Bytes=28)
3    1     TABLE ACCESS (FULL) OF 'CUSTOMERS' (Cost=360
Card=50000 Bytes=6650000)
\end{verbatim}
Correlated Subqueries

This example of a correlated subquery finds the latest promotion date for every promotion:

```sql
SQL> SELECT p1.*
      2  FROM   promotions p1
      3  WHERE  p1.promo_begin_date =
      4      (SELECT MAX(p2.promo_begin_date)
      5           FROM   promotions p2
      6           WHERE p1.promo_id = p2.promo_id);
```

The above query has the following execution plan:

```
Execution Plan
----------------------------------------------------------
0      SELECT STATEMENT Optimizer=CHOOSE (Cost=13 Card=3 Bytes=294)
1    0   HASH JOIN (Cost=13 Card=3 Bytes=294)
2    1     VIEW OF ’VW_SQ_1’ (Cost=9 Card=501 Bytes=11022)
3    2       SORT (GROUP BY) (Cost=9 Card=501 Bytes=5010)
4    3         TABLE ACCESS (FULL) OF ’PROMOTIONS’ (Cost=3 Card=501 Bytes=5010)
5    1     TABLE ACCESS (FULL) OF ’PROMOTIONS’ (Cost=3 Card=501 Bytes=38076)
```

The hash join operation in this case can be considered as access to `p1` as the outer table and the access to `p2` as the inner table.

Anti-Joins

An anti-join is a select statement with a subquery with a `NOT IN` predicate. For example:

```sql
SQL> SELECT c.*
      2  FROM   customers c
      3  WHERE  c.cust_income_level = ’F: 110,000 - 129,999’
      4  AND    c.country_id NOT IN
      5      (SELECT co.country_id
      6           FROM   countries co
      7           WHERE co.country_subregion = ’Europe’);
```

An anti-join returns rows from the left side of the predicate for which there are no corresponding rows on the right side of the predicate.

Semijoins

A semijoin returns rows that match an `EXISTS` subquery. For example:

```sql
SQL> SELECT c.*
      2  FROM   customers c
      3  WHERE EXISTS (SELECT ’x’
      4      FROM   countries co
      5      WHERE  c.country_id = co.country_id);
```
Hash Join Parameters

HASH_JOIN_ENABLED, HASH_AREA_SIZE, and HASH_MULTIBLOCK_IO_COUNT all control the way hash joins are performed.

HASH_JOIN_ENABLED
This parameter specifies whether the optimizer should consider using a hash join as a join method. (When set to FALSE, hash joining is turned off; that is, it is not available as a join method that the optimizer can consider. However, hash joining is still available on statement level—using a hint—even if the HASH_JOIN_ENABLED parameter is set to FALSE.)

HASH_AREA_SIZE
This parameter specifies the maximum amount of memory, in bytes, to be used for hash joins.

HASH_MULTIBLOCK_IO_COUNT
This parameter specifies how many sequential blocks a hash join reads and writes in one I/O.

Sort Parameters
Several sort parameters influence how the Oracle9i Server performs a sort operation. Sort performance, especially when dealing with large amounts of data, varies a great deal based on different values of these parameters. Sort performance directly influences sort/merge join performance.
Throwaway of Rows

- Rows that are retrieved but not used
- Identify throwaway by comparing the number of rows of the following:
  - The join operation
  - The input row sources

Examples

<table>
<thead>
<tr>
<th>Rows</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>NESTED LOOPS</td>
</tr>
<tr>
<td>45</td>
<td>TABLE ACCESS (...) OF our_outer_table</td>
</tr>
<tr>
<td>4530</td>
<td>TABLE ACCESS (...) OF our_inner_table</td>
</tr>
</tbody>
</table>

In this example, approximately 98% of the retrieved rows (4,430 out of 4,530 rows from the inner table) are thrown away, so there should be room for performance improvement.

<table>
<thead>
<tr>
<th>Rows</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>MERGE (JOIN)</td>
</tr>
<tr>
<td>23</td>
<td>SORT (JOIN)</td>
</tr>
<tr>
<td>23</td>
<td>TABLE ACCESS (...) OF table_A</td>
</tr>
<tr>
<td>150</td>
<td>SORT (JOIN)</td>
</tr>
<tr>
<td>150</td>
<td>TABLE ACCESS (...) OF table_B</td>
</tr>
</tbody>
</table>

In this example, only a small fraction of the retrieved rows are thrown away, so you cannot expect significant performance improvements by reducing throwaway. TKPROF is required to determine the throwaway rows because AUTOTRACE does not show this.
Minimize Throwaway of Rows

- Compare available indexes with join and nonjoin predicates.
- Consider hinting for a nested loop instead of sort/merge operations.

Minimize Throwaway or Rows

Comparing available indexes with predicates means returning to the task of finding the optimal access path for a single table access. In several situations, the data model itself is the core of a throwaway problem. Consider the following two examples.

N-Way Joins

An N-way join is a join with more than two tables, where all tables reference one another. If the references are of such a type that the fully specified access to any of the three tables requires the foreign keys from both the other tables, then the join will have a built-in throwaway of rows. To improve performance, either change the data model or experiment to find the join order with the least amount of throwaway.

Sick Join Predicates

```
SQL> SELECT ...
    2 FROM ...
    3 WHERE a.key1 = b.key1
    4 AND a.key2 = DECODE(a.key2, 0, a.key2, b.key2)
```

A sick join condition only enables the join to use the first (clean) join predicate for the join operation and then throw away the row pairs that do not satisfy the second join predicate. To improve performance, change the second predicate to:

```
4 AND (a.key2 = b.key2 OR a.key2 = 0)
```

This will not remove the throwaway of rows, but should limit it significantly.
Minimize Processing

- Changing nested loops joins into sort/merge joins can do the following:
  - Remove index access
  - Add sort overhead
- Theoretically, hash joins are the most efficient join operation.
- Cluster joins need less I/O than the corresponding nested loops join.

Minimize Processing

Changing nested loops joins into sort/merge joins can remove index access but might add significant sorting overhead.

Hash joins require less processing than the same sort/merge operation; hash joins are theoretically the fastest algorithm to join tables. However, they need appropriate tuning.

Cluster joins need less I/O than the corresponding nested loops join because the child rows that belong to one parent row are stored together in the same logical cluster block.
Summary

In this lesson, you should have learned how to:

- Use the Top-N SQL feature
- Describe available join operations
- Optimize join performance against different requirements
- Influence the join order
- Discover that tuning joins is more complicated than tuning single table statements

Summary

This lesson introduced you to Top-N SQL, performance issues with sorting, and how to optimize join statements.

There are several different join operations that Oracle9i supports:

- Nested loops join
- Sort/merge join
- Hash join
- Cluster join
- Full outer Join

The Oracle9i optimizer optimizes joins in three phases. First, the optimizer generates a list of all possible join orders. Then, for each possible join order, the best join operation is selected. Finally, the Oracle9i optimizer determines the best access path for each row source of the execution plan. RBO differs from CBO. Specifying a hint forces the CBO to be used.

Throwaway of rows occurs when both input row sources have more rows than the result of the join operation. Minimize throwaway of rows and improve performance.

Instructor Note

Workshops 3 (Joins) and 4 (Subqueries) can be done at this point, if not completed already.
Optimizer Plan Stability

Schedule: Timing     Topic
40 minutes         Lecture
30 minutes         Practice
70 minutes         Total
Objectives

After completing this lesson, you should be able to do the following:

• Identify the purpose and benefits of optimizer plan stability
• Create stored outlines
• Use stored outlines
• Edit stored outlines
• Maintain stored outlines
Optimizer Plan Stability

- Enables well-tuned applications to force the use of the desired SQL access path
- Maintains consistent execution plans through database changes
- Is implemented using stored outlines consisting of hints
- Groups stored outlines in categories

Optimizer Plan Stability

For every query, the optimizer prepares a tree of operations called an execution plan that defines the order and methods of operation that the Oracle server follows to execute the query.

Because the optimizer may have incomplete information, sometimes the best possible plan is not chosen. In these cases, you may find it worthwhile to influence the optimizer’s plan selection by rewriting the SQL statement, by using hints, or by using other tuning techniques. When satisfied, you may want to ensure that the same tuned plan is generated whenever the same query is recompiled, even when factors that affect optimization have changed.

Oracle9i provides you with a means of stabilizing execution plans across Oracle releases, database changes, or other factors that can cause an execution plan to change. You can create a stored outline containing a set of hints used by the optimizer to create an execution plan. Stored outlines can be grouped in named categories. Different categories are useful for creating groups of execution paths for different situations. The same SQL statement may have a stored outline in more than one category. For example, you may want to have an OLTP category and a DSS category.

Instructor note

Use EXPLAIN or SQL Trace and TKPROF to get the correct execution plan. In the next two pages, the CREATE_STORED_OUTLINES and USE_STORED_OUTLINES parameters work with ALTER_SYSTEM and ALTER_SESSION, but not in the init.ora file. This was done within the initiative to reduce the overall number of Oracle8i initialization parameters.
Plan Equivalence

- SQL statement text must match.
- Plans are maintained through:
  - New Oracle versions
  - New statistics on objects
  - Initialization parameter changes
  - Database reorganizations
  - Schema changes
- Plan equivalence can control execution plans for third-party applications.

SQL Statement Equivalence

Plan stability relies on exact textual matching of queries when determining whether a query has a stored outline. This is the same criterion that is used to determine whether an execution plan in the shared pool can be reused.

Stored outlines rely partially on hints that the optimizer uses to achieve stable execution plans. Therefore, the degree to which a plan remains equivalent depends on the capabilities of the hints that the plan uses. The execution steps included in a stored outline include row access methods, join order, join methods, distributed accesses, and view/subquery merging. Distributed access does not include the execution plan on the remote node.

Plan Stability

These plans are maintained through many types of database and instance changes. Therefore, if you develop applications for mass distribution, you can use stored outlines to ensure that all your customers access the same execution plans. For example, if the schema is changed by adding an index, the execution plans do not change. You must generate new stored outlines, or disable the plan stability feature, to benefit from the new index.
Creating Stored Outlines

• For all statements during a session:

```
SQL> ALTER SESSION
  2  SET CREATE_STORED_OUTLINES = TRAIN;
SQL> SELECT ... ;
SQL> SELECT ... ;
```

• For a specific statement:

```
SQL> CREATE OR REPLACE OUTLINE CU_CO_JOIN
  2  FOR CATEGORY TRAIN ON
  3      SELECT co.country_name,
  4      cu.cust_city, cu.cust_last_name
  5      FROM   countries co , customers cu
  6      WHERE  co.country_id = cu.country_id
  7      ...  
```

Creating Stored Outlines
You can create outlines for a session, or you can create them for specific SQL statements. Outlines can derive their input from both the RBO and CBO. However, the Oracle9i optimizer can only use outlines when using the CBO, because outlines rely on hints.

CREATE_STORED_OUTLINES Parameter
The Oracle server creates stored outlines automatically when you set this parameter to TRUE or to a category name. When the parameter is set to TRUE, the DEFAULT category is used. When the parameter is activated, the Oracle server creates outlines for all executed SQL statements. You can deactivate this by setting the parameter to FALSE. When you use this parameter, the Oracle server generates the names for each stored outline.

CREATE OUTLINE Command
You can also create stored outlines for specific statements by using the CREATE OUTLINE command.

One advantage of using this command is that you can name stored outlines. In the example, the CREATE OUTLINE command assigns the category TRAIN to the stored outline. If you omit a category name when generating outlines, they are placed in the DEFAULT category.

Instructor Note
Note the comments on the previous page state that it is not a good idea to enable this feature at the instance level. You usually create stored outlines at the statement or session level only. Demonstration script: demo10_01.sql
Using Stored Outlines

- **Set** USE_STORED_OUTLINES **to** TRUE 
or to a category name.

```
SQL> ALTER SESSION
    2  SET USE_STORED_OUTLINES = TRAIN;
SQL> SELECT ... 
```

- **You can set** CREATE_STORED_OUTLINES **and** 
  USE_STORED_OUTLINES **at two levels:**
  
  - ALTER SYSTEM
  
  - ALTER SESSION

---

**Using Stored Outlines**

To use a stored outline for a particular SQL statement:

- **USE_STORED_OUTLINES** must be set to TRUE or a category name:
  - If it is set to TRUE, then outlines from the DEFAULT category are used.
  - If it is set to a category name, then the outlines from that category are used. If 
    there is no matching outline in that category, but there is one in the DEFAULT 
    category, then that outline is used.

- The statement must match the text of the statement in the outline. They are compared 
  using the same method for comparing cursors in the shared pool. This means that hints 
  in the outline must be used in the statement text to cause a match. Values in bind 
  variables do not need to match.

**Instructor Note**

Demonstration script: demo10_02.sql

**Instructor Note (for page 10-7)**

The USER_OUTLINE_HINTS view also shows a STAGE column, indicating the stage 
(during compilation) at which the hint was applied.

Demonstration script: demo10_03.sql
Data Dictionary Information

Information about stored outlines can be obtained from the USER_OUTLINES and USER_OUTLINE_HINTS data dictionary views. The column descriptions of these two views are:

**USER_OUTLINES**

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Name of the outline</td>
</tr>
<tr>
<td>CATEGORY</td>
<td>User-defined name of the category to which this outline belongs</td>
</tr>
<tr>
<td>USED</td>
<td>Outline usage indication (USED, UNUSED, or UNDEFINED)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>Timestamp of outline creation</td>
</tr>
<tr>
<td>VERSION</td>
<td>Oracle version that created the outline</td>
</tr>
<tr>
<td>SQL_TEXT</td>
<td>SQL text of the query, including any hints that were part of the original statement</td>
</tr>
</tbody>
</table>

**USER_OUTLINE_HINTS**

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Name of the outline</td>
</tr>
<tr>
<td>NODE</td>
<td>ID of the query or subquery to which the hint applies (The top-level query is labeled 1; subqueries start with 2.)</td>
</tr>
<tr>
<td>JOIN_POS</td>
<td>Position of the table in the join order (The value is 0 for all hints except access method hints, which identify a table to which the hint and the join position apply.)</td>
</tr>
<tr>
<td>HINT</td>
<td>Text of the hint</td>
</tr>
</tbody>
</table>
Execution Plan Logic

To determine a SQL statement’s execution plan, Oracle9i uses the following logic:

- The statement is compared to statements in the shared pool for matching text and outline category.
- If no matching statement is found, the data dictionary is queried for a matching outline.
- If a matching outline is found, Oracle9i integrates the outline into the statement and creates the execution plan.
- If no outline is found, the statement is executed using normal (nonoutline) methods.

If an outline specifies the use of an object that cannot be used, the statement will not use the hint. For example, it references an index that no longer exists. To verify that a stored outline is being used, the explain plan for a statement must be compared when running with and without the USE_STORED_OUTLINES set.
Maintaining Stored Outlines

Use procedures in the OUTLN_PKG package to manage stored outlines and their categories:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP_UNUSED</td>
<td>Drops outlines that have not been used since they were created</td>
</tr>
<tr>
<td>DROP_BY_CAT</td>
<td>Drops outlines assigned to the specified category name</td>
</tr>
<tr>
<td>UPDATE_BY_CAT</td>
<td>Reassigns outlines from one category to another</td>
</tr>
</tbody>
</table>

You can alter individual outlines by using the ALTER OUTLINE command and drop them by using the DROP OUTLINE command.

You can export and import plans by exporting the schema OUTLN, where all outlines are stored. Outlines can be queried from tables in the OUTLN schema:

- OL$ contains the outline name, category, create timestamp, and the text of the statement.
- OL$HINTS contains the hints for the outlines in OL$.

Also, there are equivalent data dictionary views: DBA_OUTLINES and DBA_OUTLINE_HINTS.

**Note**: Because the user OUTLN is automatically created at database creation with password OUTLN, this password should be changed for security reasons.
Maintaining Stored Outlines (continued)

The DROP_UNUSED procedure does not accept any arguments and automatically drops all outlines that have never been used since they were created.

The UPDATE_BY_CAT procedure example moves all outlines from the DEFAULT category to the TRAIN category. For example, this procedure is useful to move a set of outlines from an experimental (test environment) category into a production category.

The DROP_BY_CAT procedure example purges all outlines that belong to the TRAIN category in a single call.

Export and Import Stored Outlines

By exporting and importing the OUTLN schema, you can distribute outlines to other databases, such as to all departments that use a certain application. This approach ensures execution plan consistency at all sites.

Instructor Note

Demonstration script: demo10_04.sql, then run demo10_03.sql again.
Outline Editing Overview

- Stored outline editing is new in Oracle9i.
- Tune execution plans without having to change the application.
- This is possible by editing the content of the saved plan.

Outline Editing Overview

In Oracle9i, outline editing extends the usefulness of stored outlines by introducing a user-editing interface, which enables users and third-party vendors to tune their execution plans by editing the stored outlines that are used to influence the optimizer. This can be useful when using the same application in different environments (small versus big databases).

This feature benefits both application developers and customer support personnel. Although the optimizer usually chooses optimal plans for queries, there are times when the user knows something about the execution environment that is inconsistent with the heuristics the optimizer follows. Sometimes an execution plan is acceptable in one environment but not in another. By editing the outline directly, the user can tune the query without having to change the application.

The application developer might generate outlines in a staging area and notice that some plan did not take advantage of an index that could improve performance. It might be easier to edit the outline to use the index rather than searching through the application code and tuning the SQL until it eventually yields the desired result.

Customer support might be at a customer site investigating a problem query. By creating an outline and then editing it, it is likely that with some simple edits, such as changing join order, the problem can be solved quickly. In this way, the customer’s problem is solved immediately without having to go through the process of debugging and updating the application itself.
Outline Editing Overview

- The outline is cloned in a staging area
- The outline is edited in the user’s session
- When satisfied with the result, the editor can publicize the result to the user community.

Outline Editing Overview (continued)

For the customer whose environment has unique characteristics that might cause an outline to yield a less than optimal execution plan, the ability to make minor adjustments to the outline enhances the ability to support specific customer needs. In this sense, stored outlines are made more adaptive as users can make finely tuned adjustments to the saved plan. Stored outline metadata is maintained in the OUTLN schema and maintained by the server. You are advised not to update these tables directly in the same way you are advised not to update system tables. Therefore, Oracle must provide users with a way to safely edit an outline without compromising the integrity of the outline for the rest of the user community.

To accomplish this, Oracle proposes that the outline be cloned into the user’s schema at the onset of the outline editing session. All subsequent editing operations are performed on that clone until the user is satisfied with the edits and chooses to publicize them. In this way, any editing done by the user does not impact the rest of the user community, which continues to use the public version of the outline until the edits are explicitly saved.
Editable Attributes

- Join order
- Join methods
- Access methods
- Distributed execution plans
- Distribution methods for parallel query execution
- Query rewrite
- View and subquery merging

**Join order**: Join order defines the sequence in which tables are joined during query execution. This includes tables produced by evaluating subqueries and views as well as tables appearing in the FROM clauses of subqueries and views.

**Join methods**: Join methods define the methods used to join tables during query execution. Examples are nested loops join and sort/merge join.

**Access methods**: Access methods define the methods used to retrieve table data from the database. Examples are indexed access and full table scan.

**Distributed execution plans**: Distributed queries have execution plans that are generated for each site at which some portion of the query is executed. The execution plan for the local site at which the query is submitted can be controlled by plan stability and equivalent plans must be produced at that site. In addition, driving site selection can be controlled centrally even though it might normally change when certain schema changes are made.

**Distribution methods**: For parallel query execution, distribution methods define how the inputs to execution nodes are partitioned.

**View and subquery merging and summary rewrite**: View and subquery merging and summary rewrite is meant to include all transformations in which objects or operations that occur in one subquery of the original SQL statement are caused to migrate to a different subquery for execution. Summary rewrite can also cause one set of objects or operations to be replaced by another.
Outline Cloning

• Public outlines:
  – Default setting when creating outlines
  – Stored in the OUTLN schema
  – Used when USE_STORED_OUTLINES is set to TRUE or a category

• Private outlines:
  – Stored in the user’s schema for the duration of the session
  – Can be edited
  – Used when USE_PRIVATE_OUTLINES is set to TRUE or a category
  – Changes can be saved as public outlines

Outline Cloning

Public Outlines

In Oracle9i, all outlines are public objects indirectly available to all users on the system for whom the USE_STORED_OUTLINES configuration parameter setting applies. Outline data resides in the OUTLN schema that can be thought of as an extension to the SYS schema, in the sense that it is maintained by the system only. You are discouraged from manipulating this data directly to avoid security and integrity issues associated with outline data. Outlines will continue to be public by default and only public outlines are generally available to the user community.

Private Outlines

In Oracle9i, the notion of a private outline to aid in outline editing is introduced. A private outline is an outline seen only in the current session and whose data resides in the current parsing schema. By storing the outline data for a private outline directly in the user’s schema, users are given the opportunity to manipulate the outline data directly through DML in whichever way they choose. Any changes made to such an outline are not seen by any other session on the system and applying a private outline to the compilation of a statement can be done only in the current session through a new session parameter. Only when a user explicitly chooses to save edits back to the public area do the rest of the users see them.

An outline clone is a private outline that has been created by copying data from an existing outline.
Outline: Administration and Security

- Privileges required for cloning outlines:
  - SELECT_CATALOG_ROLE
  - CREATE ANY OUTLINE
- DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES
  - Creates required temporary tables in user’s schema for cloning and editing outlines
  - Requires EXECUTE privilege on DBMS_OUTLN_EDIT

Outline: Administration and Security

SELECT_CATALOG_ROLE

This role is required for the CREATE OUTLINE FROM command unless the issuer of the command is also the owner of the outline. Any CREATE OUTLINE statement requires the CREATE ANY OUTLINE privilege. Specification of the FROM clause will require the additional SELECT_CATALOG_ROLE role because such a command exposes SQL text to different users who might otherwise not be privileged to read the text.

DBMS_OUTLN_EDIT.CREATE_EDIT_TABLES

A supporting command procedure that creates the metadata tables in the invoker’s schema. This procedure is callable by anyone with the EXECUTE privilege on the DBMS_OUTLN_EDIT package. Refer to the Supplied PL/SQL Packages Reference for more information. Note also that the DBMS_OUTLN package is synonymous with OUTLN_PKG.

Private outlines are private to the session editing them. To accommodate the possibility of multiple users editing the same outline at the same time (not a good practice, but possible) the data is partitioned by session. This is really scratch data not intended for long-term use and this removes the need for users to worry about cleaning up when they are done because the temporary data goes away when the session is closed.

Oracle9i: SQL Tuning Workshop 10-15
Outline: Administration and Security (continued)

V$SQL

A column is added to the V$SQL fixed view to help users distinguish whether a shared cursor was compiled while using a private outline or a public outline. OUTLINE_SID is the name of this new column and it identifies the session ID from which the outline was retrieved. The default is 0 which implies a lookup in the OUTLN schema.
Configuration Parameters

USE_PRIVATE_OUTLINES is a session parameter that controls the use of private outlines instead of public outlines.

```sql
ALTER SESSION SET USE_PRIVATE_OUTLINES =
    TRUE | FALSE | category_name ;
```

- **TRUE** enables the use of private outlines in the DEFAULT category.
- **FALSE** disables use of private outlines.
- **category_name** enables use of private outlines in the named category.

Configuration Parameters

The USE_PRIVATE_OUTLINES session parameter is added to control the use of private outlines instead of public outlines. When an outlined SQL command is issued, this parameter causes outline retrieval to come from the session private area rather than the public area that is usually consulted as per the setting of USE_STORED_OUTLINES. If no outline exists in the session private area, no outline is used for the compilation of the command.

You can specify a value for this session parameter by using the following syntax:

```sql
ALTER SESSION SET USE_PRIVATE_OUTLINES =
    TRUE | FALSE | category_name ;
```

Where:

- **TRUE** enables use of private outlines and defaults to the DEFAULT category
- **FALSE** disables use of private outlines
- **category_name** enables use of private outlines in the named category

When a user begins an outline editing session, the parameter should be set to the category to which the outline being edited belongs. This enables the feedback mechanism in that it allows the private outline to be applied to the compilation process.

Upon completion of outline editing, this parameter should be set to **FALSE** to restore the session to normal outline lookup as dictated through the USE_STORED_OUTLINES parameter.
Create Outline Syntax

The **CREATE OUTLINE** command has been extended to clone outlines:

```
CREATE [OR REPLACE]
    [PUBLIC | PRIVATE] OUTLINE [outline_name]
    [FROM [PUBLIC | PRIVATE] source_outline_name]
    [FOR CATEGORY category_name] [ON statement]
```

Create Outline Syntax Changes

The new syntax elements in the above slide are:

**PUBLIC**: The outline is to be created for use by PUBLIC. This is the default because outline creation is intended for systemwide use.

**PRIVATE**: The outline is to be created for private use by the current session only and its data is stored in the current parsing schema. When specified, the prerequisite outline tables and indices must exist in the local schema.

**FROM**: This construct provides a way to create an outline by copying an existing one.

**source_outline_name**: This is the name of the outline being cloned. By default, it is found in the public area, but if preceded by the PRIVATE keyword it will be found in the local schema.

The addition of the PRIVATE and FROM keywords enable outline cloning. When you want to edit an outline, you do so on a private copy which is created by specifying the PRIVATE keyword. In the FROM clause, the source outline to be edited is named and is found in the public area unless preceded by the PRIVATE keyword, in which case the you would be copying a private version of the named outline.

When specifying the FROM clause, existing semantics apply to the outline name and category, so if unspecified, an outline name is generated under the DEFAULT category. When a PRIVATE outline is being created, if the prerequisite outline tables that hold the outline data do not exist in the local schema, an error is returned.

*Oracle9i: SQL Tuning Workshop 10-18*
Outline Cloning Examples

• Clone a public outline to the session private area:

```
CREATE PRIVATE OUTLINE private_outline FROM public_outline;
```

• Clone a private copy to another private copy:

```
CREATE PRIVATE OUTLINE private_outline2 FROM PRIVATE private_outline1 FOR CATEGORY cat2;
```

• Copy a private outline back to the public area:

```
CREATE OR REPLACE OUTLINE public_outline FROM PRIVATE private_outline
```

Outline Cloning Examples

```
CREATE PRIVATE OUTLINE private_outline
FROM public_outline;
```

This example shows how to clone a public outline to the session private area for editing purposes. In this case, PUBLIC_OUTLINE is cloned into the private area and given the name PRIVATE_OUTLINE.

```
CREATE PRIVATE OUTLINE private_outline2
FROM PRIVATE private_outline1 FOR CATEGORY cat2;
```

This example shows how to clone a private copy of an outline to another private copy. In this case, PRIVATE_OUTLINE2 is being created in the CAT2 category from the PRIVATE_OUTLINE1 outline, which is local to the current session. This might be done if the user wants to save several versions of the outline. Specification of the category is necessary here to avoid namespace collision.

```
CREATE OR REPLACE OUTLINE public_outline
FROM PRIVATE private Outline
```

This example shows how to copy a private outline back to the public area for general use. This is a way to save local edits back to the master copy of the outline. This operation is also called publicize.
Outline Cloning Examples

- Create a private outline independently:
  ```sql
  CREATE PRIVATE OUTLINE private_outline
  ON select count(*) from customers;
  ```

- Copy one public outline to another:
  ```sql
  CREATE OR REPLACE OUTLINE public_outline2
  FROM public_outline1 FOR CATEGORY cat2;
  ```

Outline Cloning Examples (continued)

```sql
CREATE PRIVATE OUTLINE private_outline
ON select count(*)
from customers;
```

This example demonstrates how you might create a private outline independently without copying from a preexisting outline. PRIVATE_OUTLINE is created in the session private area. This is useful for users who want to experiment with an outline before creating it for public consumption.

```sql
CREATE OR REPLACE OUTLINE public_outline2
FROM public_outline1 FOR CATEGORY cat2;
```

This example shows how to copy one public outline to another. Here the PUBLIC_OUTLINE1 public outline is copied to PUBLIC_OUTLINE2 and placed in the CAT2 category. Specification of the category is necessary here to avoid namespace collision.
Outline Cloning Examples

Replace an outline with itself:

```sql
CREATE OR REPLACE PRIVATE OUTLINE priv_outll
FROM PRIVATE priv_outll;
```

```sql
dbms_outln_edit.refresh_private_outline(
  'private_outll1')
```

Outline Cloning Examples (continued)

```sql
CREATE OR REPLACE PRIVATE OUTLINE private_outll1
FROM PRIVATE private_outll1;
```

This example shows how to replace an outline with itself. This might seem like a useless operation, but it actually serves the dual purpose of invalidating dependent shared cursors and bringing the shared pool view of the object in line with what is on disk. The feedback mechanism relies on this aspect of the command. You can also use `DBMS_OUTLN_EDIT.REFRESH_PRIVATE_OUTLINE` or `ALTER SYSTEM FLUSH SHARED_POOL` to accomplish this.
Summary

In this lesson, you should have learned how to:

- Use stored outlines to ensure execution plan consistency
- Create outlines for a session or a single statement
- Organize outlines in categories
- Enable or disable using outlines or categories of outlines
- Maintain outlines with the OUTLN_PKG package or the ALTER OUTLINE command

Summary

This lesson introduced you to the use of stored outlines and its influence on optimizer plan stability, and use of the package OUTLN_PKG on stored outlines.

Oracle9i provides you with a means of stabilizing execution plans across Oracle releases, database changes, or other factors that can cause an execution plan to change. You can create a stored outline containing a set of hints used by the optimizer to create an execution plan.

Stored outlines rely partially on hints that the optimizer uses to achieve stable execution plans. These plans are maintained through many types of database and instance changes. You can use stored outlines to ensure that all your customers access the same execution plans.

You can create outlines for a session, or you can create them for specific SQL statements.

To use stored outlines, the USE_STORED_OUTLINES option must be set to TRUE or a category name. Use procedures in the OUTLN_PKG package to manage stored outlines and their categories. There are data dictionary views that store information on the outlines.

Note: For more details about plan stability and using stored outlines, refer to Oracle9i Tuning, “Optimizer Modes, Plan Stability, and Hints.”
Practice Overview

This practice covers the following topics:

- Creating a stored outline
- Altering your session to use a stored outline
- Using a stored outline
- Querying the data dictionary
- Removing a stored outline

Practice Overview

In this practice, you create, use, and then remove a stored outline. You must alter your session in order for the outline to be used. Use the USER_OUTLINES and USER_OUTLINE_HINTS data dictionary views to view the status of your outline.
Practice 10

1. Turn SQL*Plus AUTOTRACE off. Make sure that the SUPPLIER_ID column of the PRODUCTS table is not indexed by running the dai.sql script from the demo labs directory, then create a stored outline for the following SQL statement:
   ```sql
   SQL> @dai.sql
   SQL> select p.prod_id, p.prod_name, p.prod_min_price
   2   from products p
   3   where p. supplier_id = 260;
   ```

2. Check the USER_OUTLINES data dictionary view to find the outline name and category name for the outline you just created; also, query the USED column to see that the outline is not yet used.

3. Check the USER_OUTLINE_HINTS data dictionary view to see which hints are stored with your outline.

4. Issue an ALTER SESSION command to enable your session using the stored outline.

5. Execute the SQL statement in step 1, create an index on the PROD_STATUS column in the PRODUCTS table, and execute the SQL statement again to see whether the execution plan has changed.

6. Check the USER_OUTLINES data dictionary view again to see whether the outline has been used.

7. Drop the outline, and issue the SQL statement again to see that the index will now be used.
Advanced Indexes

Schedule: Timing   Topic

45 minutes   Lecture
45 minutes   Total

Instructor Note

- Note that index hints are revisited, adding the INDEX_COMBINE hint to the hints covered in the “Influencing the Optimizer” lesson. Star transformation is covered in this lesson because it is a feature based on bitmapped indexes. Make a clear distinction between star transformation (as discussed in this lesson) and star joins (as discussed in the “Sorting and Joining” lesson). Finally, function-based indexes are introduced.

- To speed the demonstrations against the SALES table, create a copy of it using a subset of rows from the original SALES table.

To create the copy of the SALES table, execute the command (this command is also provided in the demo11_01.sql script):

```
CREATE TABLE copy_sales AS
SELECT *
FROM   sales
WHERE  rownum < 200000;
```
Objectives

After completing this lesson, you should be able to do the following:

- Create bitmapped indexes
- Identify bitmapped index operations
- Specify bitmapped index hints
- Use star transformations
- Create function-based indexes
- View data dictionary information

Objectives

After completing this lesson, you should be able to:

- Create bitmapped indexes, identify bitmapped index operations, and use bitmapped index hints.
- Describe, identify and use star transformations.
- Create and use function-based indexes.
- View data in the data dictionary on indexes.
Bitmapped Indexes

- Compared with regular B*-tree indexes, bitmapped indexes are faster and use less space for low cardinality columns.
- Each bitmap index comprises storage pieces called bitmaps.
- Each bitmap contains information about a particular value for each of the indexed columns.
- Bitmaps are compressed and stored in a B*-tree structure.

Bitmapped Indexes

Bitmap indexing provides both substantial performance improvements and space savings over usual (B*-tree) indexes when there are few distinct values in the indexed columns.

If the number of distinct keys in a column is less than or equal to 1% of the total number of rows for a table, then the column has low cardinality. For example, in a table with 10,000 rows, if there are fewer than 100 different values in a particular column, then that column has low cardinality, and a bitmap index may provide the best performance for queries based on that column.

The maximum number of columns in a single bitmapped index is 30.

The Oracle9i Server compresses bitmap storage, making bitmapped indexes very storageefficient. The bitmaps are stored in a B*-tree structure internally to maximize access performance.

Note: Rule-based optimization does not consider using bitmapped indexes.
**Bitmapped Index Structure**

Each position in a bitmap stores information about a particular row.

```
0  →  Row #1
1  →  Row #2
1  →  Row #3
0 
1 
1 
0 
```

Any type of index is designed to help find rows that contain certain key values. A regular (B*-tree) index does this by pairing the ROWID for a row with its key value, then sorts those pairs into a B*-tree structure. The ROWID serves as a pointer to a row with that value.

A bitmapped index has a very different structure; no ROWIDs are stored. Because each different key value has its own bitmap, you benefit the most from bitmapped indexes when you only have a few distinct index values.

Each bitmap header stores start and end ROWIDs. Based on these values, the Oracle9i Server uses an internal algorithm to map bitmaps onto ROWIDs.

Each position in a bitmap maps to a potential row in the table. The contents of that position in the bitmap for a particular value indicate whether that row has that value in the bitmapped columns. The value stored is a 1 if the row values match the bitmap condition and 0 otherwise.
Creating Bitmapped Indexes

To create a bitmapped index, use the `CREATE INDEX` command and add the keyword `BITMAP`. In the above example, a bitmap index is being created on the `SUPPLIER_ID` column of the `PRODUCTS` table.

The Oracle9i Server then creates a series of bitmaps, each one corresponding to a particular value of the columns involved. In the example, if the `SUPPLIER_ID` column has 289 distinct values, then 289 bitmaps are built. If another `SUPPLIER_ID` value were later added, a new bitmap for that value is created.

In the bitmap for `SUPPLIER_ID '1'`, the first position contains a one, showing that the first row has that value. In the bitmap for `SUPPLIER_ID '2'`, the first position contains a zero because the first row does not have that value. Each row maps to a position in the bitmaps, and the one matching its conditions will contain the digit one (1) in that position.

If a bitmap index involves more than one column, then there will be a bitmap for every possible combination.

**Note:** Bitmapped indexes cannot be declared as unique (B*-tree indexes are more appropriate in such cases anyway), and neither can the `NOSORT` option be used when creating bitmapped indexes.
Creating Bitmapped Indexes

If the table is partitioned, you must use the LOCAL option to create the index:

```
CREATE BITMAP INDEX index_name
ON table_name(column_name)
LOCAL [NOLOGGING|LOGGING COMPUTE STATISTICS];
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL</td>
<td>Must be a locally partitioned bitmap index</td>
</tr>
<tr>
<td>LOGGING</td>
<td>NOLOGGING</td>
</tr>
<tr>
<td>COMPUTE STATISTICS</td>
<td>Collects statistics at relatively little cost during the creation of an index</td>
</tr>
</tbody>
</table>
Example of Querying on a Bitmapped Index

A query that includes a WHERE condition involving the bitmapped column can use the bitmapped index. In the above example, a search is being conducted for SUPPLIER_ID of 3. This query may scan the bitmap for SUPPLIER_ID ‘3’ for positions with a 1 in them. Positions with a 1 will have their corresponding rows returned for the query. In some cases, such as a query counting the number of rows with SUPPLIER_ID ‘3’, the query might simply use the bitmap itself and count the number of 1s, not needing the actual rows.

Instructor Note

You can discuss the benefit of a bitmapped index in this single predicate statement. Note that this example is meant to illustrate bitmapped indexes only. Usually, bitmapped indexes are efficient with more complex WHERE clauses using AND/OR/NOT.
Combining Bitmapped Indexes

Due to fast bit-and, bit-minus, bit-or operations, bitmapped indexes are efficient:

- When using IN (value_list)
- When predicates are combined with AND/OR

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A or B</th>
<th>not A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Combining Bitmapped Indexes

You can use bitmapped indexes efficiently when a query combines several possible values for a column or when two separately indexed columns are used.

Query Combining Multiple Values for a Column

In some cases, a WHERE clause might allow several different values for a column:

```
SQL> SELECT * FROM products
  2  WHERE supplier_id IN (1, 3);
```

In this case, the bitmaps for both 1 and 3 would be used. For each position, if either of the bitmaps has a 1 in that position, then the row is returned. A bit-or operation is performed first, followed by a translation into ROWIDs.

Query Combining Two Different Bitmapped Indexes

In some cases, a WHERE clause might reference several separately indexed columns.

```
SQL> SELECT * FROM products
  2  WHERE supplier_id = 3
  3   AND prod_pack_size = 'card box';
```

If both the SUPPLIER_ID and PROD_PACK_SIZE columns have bitmapped indexes, a bit-and operation on the two bitmaps would quickly locate the rows that you are looking for. The more complex the compound WHERE clauses get, the more you benefit from bitmapped indexing.
When to Use Bitmapped Indexes

- Columns with low cardinality
- Columns that are frequently used in:
  - Complex WHERE clause conditions
  - Group functions (such as COUNT, SUM)
- Very large tables
- DSS systems with many ad hoc queries and few concurrent DML changes

When to Use Bitmapped Indexes

Use bitmapped indexes in the following circumstances:

- The column has a low cardinality, meaning that there are few possible values for the column. A slightly higher cardinality would also be appropriate if the column is frequently used in complex conditions in the WHERE clauses of queries.
- The WHERE clauses of queries contain multiple predicates of these low- or medium-cardinality columns. Bitmapped indexes are especially helpful for complex ad hoc queries with lengthy WHERE clauses or aggregate queries (containing SUM, COUNT, or other aggregate functions).
- The table has many rows. On a table with 1 million rows, even 10,000 distinct possible values would be acceptable.
- There are frequent, possibly ad hoc, queries on the table.
- The environment is oriented toward data warehousing (decision support systems).

Bitmapped indexes are not ideal for OLTP environments because of their locking behavior. It is not possible to lock a single bitmap position. The smallest amount of a bitmap that can be locked is a bitmap segment, which can be up to half a data block in size. Changing the value of a row results in a bitmap segment becoming locked, in effect blocking changes on a number of rows. This is a serious disadvantage when there are many UPDATE, INSERT, or DELETE statements being issued by users. It is not a problem when data is loaded or updated in bulk actions, as in data warehousing environments.
Advantages of Bitmapped Indexes

When used appropriately, bitmapped indexes provide:

- Reduced response time for many ad hoc queries
- Substantial reduction of space usage compared to other indexing techniques
- Dramatic performance gains, even on low-end hardware

Advantages of Bitmapped Indexes

When used appropriately, bitmap indexes can provide substantial advantages:

- They provide reduced response time for many ad hoc queries. The cost-based optimizer is able to consider the use of bitmapped indexes and decide when their use would be the most efficient way to execute a given query, incorporating parallelism, if appropriate.

- They provide a substantial reduction of space usage compared to regular B*-tree indexes. If only regular (B*-tree) indexes are used, then the index designer must predict which columns will be used together and create a multicolumn (concatenated) index for these columns. This regular index would require a large amount of space and be sorted, making it useless for many queries that involve only a subset of the columns. The index designer could also create indexes on the other permutations of these columns; this would use a great deal of storage space. Bitmapped indexes, however, can be created individually and then combined efficiently at run time. If the column to be indexed is a unique key column, then the space required for the bitmapped index will exceed that of the B*-tree index. Use bitmapped indexes on low- or medium-cardinality columns only.

- You can obtain dramatic performance gains, even on low-end hardware. You do not need parallel processing capability to achieve performance gains with bitmapped indexes.
Bitmapped Index Guidelines

- **Reduce bitmap storage by:**
  - Declaring columns *NOT NULL* when possible
  - Using fixed-length data types when feasible
  - Using the `ALTER TABLE ... MINIMIZE RECORDS_PER_BLOCK` command

- **Improve bitmap performance by increasing the following parameters:**
  - `CREATE_BITMAP_AREA_SIZE` (default 8 MB)
  - `BITMAP_MERGE_AREA_SIZE` (default 1 MB)

You can do several things to improve performance and storage of bitmapped indexes:
- Declare *NOT NULL* constraints on all possible columns
- Use fixed length data types
- Use the `ALTER TABLE ... MINIMIZE RECORDS_PER_BLOCK` command

The `ALTER TABLE` command determines the number of rows that are stored per block, and limits the number of rows per block accordingly. Issue this statement *before* creating any bitmapped indexes.

The `CREATE_BITMAP_AREA_SIZE` parameter determines the amount of memory allocated for bitmap creation. The default value is 8 MB, which is enough in most cases. A larger value can result in larger continuous bitmaps and thus faster query processing. The `SORT_AREA_SIZE` parameter also influences bitmap creation performance.

The `BITMAP_MERGE_AREA_SIZE` parameter determines the amount of memory used to merge bitmaps retrieved from an index range scan. The default value is 1 MB.

**Instructor Note**

When you create a bitmapped index, the theoretical maximum number of rows per block is calculated, based on data dictionary information. When columns are *NOT NULL* and fixed in width, the row size is more predictable. This means that the bitmaps do not contain many unnecessary (zero) bits. Note that `ALTER TABLE ... MINIMIZE RECORDS_PER_BLOCK` can waste space, because it limits the number of rows stored per data block, regardless of the block space they occupy.
What Is a Bitmap Join Index?

In addition to a bitmap index on a single table, you can create a bitmap join index in Oracle9i. A bitmap join index is a bitmap index for the join of two or more tables. A bitmap join index is a space-efficient way of reducing the volume of data that must be joined by performing restrictions in advance.

As you can see in the above slide, Oracle9i introduces a new `CREATE BITMAP INDEX` syntax that you can use to specify a `FROM` and a `WHERE` clause. Here, you create a new bitmap join index named `CUST_SALES_BJI` on the `SALES` table. The key of this index is the `CUST_CITY` column of the `CUSTOMERS` table.

This example assumes that there is an enforced primary key-foreign key relationship between `SALES` and `CUSTOMERS` to ensure that what is stored in the bitmap reflects the reality of the data in the tables. The `CUST_ID` column is the primary key of `CUSTOMERS` and also the foreign key from the `SALES` table to the `CUSTOMERS` table. The `FROM` and `WHERE` clause in the `CREATE` statement allow Oracle9i to make the link between the two tables. They represent the natural join condition between the two tables.
What Is a Bitmap Join Index? (continued)

The middle part of the previous graphic shows you a theoretical implementation of this bitmap join index. Each entry or key in the index represents a possible city found in the CUSTOMERS table. A bitmap is then associated to one particular key. The meaning of the bitmap is quite obvious because it is the same representation as for traditional bitmap indexes. Each bit in a bitmap corresponds to one row in the SALES table. If you take the first key above (Rognes), you can see that the first row in the SALES table corresponds to a product sold to a Rognes customer, whereas the second bit is not a product sold to a Rognes customer.

The interesting aspect of this structure becomes clear with the last part of the slide. Indeed, when the user tries to find what the total cost of all sales for the Aix-en-Provence customers is, Oracle9i can use the above bitmap join index and the SALES table to answer the question. In this case, there is no need to compute the join between the two tables explicitly. Using the bitmap join index in this case is much faster than computing the join at query time.
Bitmap Join Index:
Advantages and Disadvantages

• Advantages:
  – Good performance for join queries and space efficient
  – Is especially useful for large dimension tables in star schemas

• Disadvantages:
  – More indexes are required: Up to one index per dimension table column rather than one index per dimension table.
  – Maintenance costs are higher: Building or refreshing a bitmap join index requires a join.

Advantages and Disadvantages

A bitmap join index is an index on one table that involves columns of one or more different tables through a join.

The volume of data that must be joined can be reduced if bitmap join indexes are used as joins that have already been precalculated. In addition, bitmap join indexes that can contain multiple dimension tables can eliminate bitwise operations that are necessary in the star transformation’s use of bitmap indexes.

An alternative to a bitmap join index is a materialized join view, which is the materialization of a join in a table. Compared to a materialized join view, a bitmap join index is much more space efficient because it compresses ROWIDs of the fact tables.

Also, queries using bitmap join indexes can be sped up by means of bitwise operations.

On the other hand, you may need to create more bitmap join indexes on the fact table to satisfy the maximum number of different queries. This mean that you may have to create one bitmap join index for each column of the corresponding dimension tables. Of course, the implication of having many indexes on one table is that maintenance costs are higher, especially when the fact table is updated.

Note: Dimensions are covered in detail in lesson 12
Advantages and Disadvantages (Continued)

Because of the necessity of storing the results of join results, bitmap join indexes have the following restrictions:

- Parallel DML is currently supported only on the fact table. Parallel DML on one of the participating dimension tables will mark the index as unusable.
- Only one table can be updated concurrently by different transactions when using the bitmap join index.
- No table can appear twice in the join.
- You cannot create a bitmap join index on an index-organized table or a temporary table.
- The columns in the index must all be columns of the dimension tables.
- The dimension table join columns must be either primary key columns or have unique constraints.
- If a dimension table has composite primary key, each column in the primary key must be part of the join.
- Bitmap join indexes cannot be built or rebuilt online.
Indexes and Row-Access Methods

- Access methods discussed in an earlier lesson:
  - B*-tree index access
  - Table access by ROWID
  - B*-tree index merge
  - Fast full index scans
- Index hints
- Star transformation
## Index Hints

Most of the following index hints are covered in the “Influencing the Optimizer” lesson:

<table>
<thead>
<tr>
<th>Index Hints</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX(t [idx]...)</td>
<td>Chooses an (ascending) index scan for the specified table</td>
</tr>
<tr>
<td>INDEX_ASC(t [idx]...)</td>
<td>Chooses an ascending index scan for the specified table</td>
</tr>
<tr>
<td>INDEX_DESC(t [idx]...)</td>
<td>Chooses a descending index scan for the specified table</td>
</tr>
<tr>
<td>AND_EQUAL(t idx idx [idx]...)</td>
<td>Chooses an execution plan that uses an access path, merging the scans on several single-column indexes (minimum 2, maximum 5)</td>
</tr>
<tr>
<td>INDEX_COMBINE(t [idx]...)</td>
<td>Uses a Boolean combination of bitmapped indexes</td>
</tr>
<tr>
<td>INDEX_FFS(t [idx]...)</td>
<td>Causes a fast full index scan to be performed</td>
</tr>
<tr>
<td>NO_INDEX(t [idx]...)</td>
<td>Explicitly disallows a set of indexes for the specified table</td>
</tr>
</tbody>
</table>
**INDEX_COMBINE** Hint Example

```sql
SQL> SELECT --+INDEX_COMBINE(SALES)
2         *
3  FROM  SALES
4  WHERE (promo_id = 9999 AND
5         cust_id <> 50)
6  OR     prod_id BETWEEN 1000 AND 10;
```

**The INDEX_COMBINE Hint**

The **INDEX_COMBINE** hint is designed for bitmapped index operations. Remember the following:

- If certain indexes are given as arguments for the hint, the optimizer tries to use some combination of those particular bitmap indexes.
- If no indexes are named in the hint, all indexes are considered hinted.
- The optimizer always tries to use hinted indexes, whether or not it considers them cost effective.

**INDEX_COMBINE** Hint Example

Suppose all three columns referenced in the **WHERE** predicate of the statement above (**PROMO_ID**, **CUST_ID**, and **PROD_ID**) have a bitmapped index. When you enable **AUTOTRACE**, the execution plan of the statement above might look as shown in the slide on the next page.

**Instructor Note**

Demonstration script: `demo11_01.sql`

Make sure that you create bitmapped indexes on the **PROMO_ID**, **CUST_ID**, and **PROD_ID** columns of the **SALES** table first. You can run the `cbi.sql` script to create the bitmapped indexes. Note that the execution plan is displayed in the next slide.
INDEX_COMBINE Hint Example

In the example, the following bitmap row sources are used:

<table>
<thead>
<tr>
<th>Bitmap Row Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITMAP CONVERSION TO ROWIDS:</td>
<td>Converts bitmaps into ROWIDs to access a table</td>
</tr>
<tr>
<td>COUNT:</td>
<td>Returns the number of entries if the actual values are not needed</td>
</tr>
<tr>
<td>BITMAP OR</td>
<td>Computes the bit-wise OR of two bitmaps</td>
</tr>
<tr>
<td>BITMAP MINUS</td>
<td>Subtracts the bits of one bitmap from another (This operator is used for negated predicates.)</td>
</tr>
<tr>
<td>BITMAP INDEX SINGLE VALUE</td>
<td>Looks up the bitmap for a single key value</td>
</tr>
<tr>
<td>RANGE SCAN:</td>
<td>Retrieves bitmaps for a value range</td>
</tr>
<tr>
<td>BITMAP MERGE</td>
<td>Merges several bitmaps resulting from a range scan into one</td>
</tr>
<tr>
<td></td>
<td>(using a bit-wise AND operator)</td>
</tr>
</tbody>
</table>
Star Transformation

One type of data warehouse design centers around what is known as a star schema, which is characterized by one or more large fact tables that contain the primary information in the data warehouse and a number of much smaller dimension tables (or lookup tables), each of which contains information about the entries for a particular attribute in the fact table.

A star query is a join between a fact table and a number of lookup tables. Each lookup table is joined to the fact table using a primary-key to foreign-key join, but the lookup tables are not joined to each other.

Star queries are quite challenging to tune. There are two basic approaches to this problem:

- Do as smart a job as possible. The “Sorting and Joining” lesson discussed how to use star joins to optimize star queries.
- Transform the statement into an easier one. This lesson discusses this approach, using bitmapped indexes. This approach is also known as star transformation.
Star Transformation

• Execute star queries efficiently, especially in the following cases:
  – Many dimension tables
  – A sparse fact table
  – When not all dimensions have constraining predicates
• Set the STAR_TRANSFORMATION_ENABLED initialization parameter
• Use the STAR_TRANSFORMATION hint

Star Transformation (continued)

Star transformation is a cost-based query transformation aimed at executing star queries efficiently.
The star transformation is especially effective if any of the following conditions exist:
  • The number of dimension tables is large; this makes the Cartesian product of the dimension tables too expensive.
  • The fact table is sparse.
  • There are queries where not all dimension tables have constraining predicates (no additional nonjoin predicates).

The transformation can choose to combine bitmapped indexes corresponding precisely to the constrained dimensions and does not need to produce the Cartesian product of the dimension tables, which is the basis of the other approach to tuning star queries.

The STAR_TRANSFORMATION_ENABLED parameter is dynamic and can be set to TRUE at the session level with the ALTER SESSION command:

```
SQL> alter session set star_transformation_enabled = true;
```
Alternatively, you can specify the STAR_TRANSFORMATION hint at the statement level.

Oracle9i: SQL Tuning Workshop 11-21
Star Transformation Example

In this example, the SALES table is the fact table, and CHANNELS, PRODUCTS, and CUSTOMERS are the dimension tables.

Because there are bitmapped indexes on the CHANNEL_ID, PROD_ID, and CUST_ID columns of the SALES table, the query can be rewritten so that these bitmapped indexes can be used.

Note that you are joining four tables but selecting only column values of three of them; the CUSTOMERS table is used only to select customers that are in Asten. Also note that there are no nonjoin predicates on the facts table.

Note: It is not possible to force star transformation, and there are several constraints. Often the optimizer does not consider a star transformation because a better execution plan is available, or one of the constraints for star transformation is violated. For more information about star transformation, see Oracle9i Concepts, “The Optimizer.”

Instructor Note

Do not try to demonstrate star transformation unless you are certain that it will work. Note that the STAR_TRANSFORMATION hint works only if the optimizer thinks it is feasible.

The SQL statement is available in the demo11_02.sql script. demo11_02a.sql contains the statement shown on the next page. The results are the same for both queries.

The next page shows how this statement can be transformed by the optimizer to benefit from bitmapped indexes. Note that this is not how it exactly works internally, but is a way to explain and visualize star transformation.
Star Transformation Example

```sql
SELECT s.amount_sold, p.prod_name, ch.channel_desc
FROM sales s, products p, channels ch
WHERE s.prod_id = p.prod_id
AND s.channel_id = ch.channel_id
AND s.cust_id in
(SELECT cust_id
FROM customers)
AND s.channel_id in
(SELECT channel_id
FROM channels
WHERE channel_id in ('I','P','S'))
AND s.cust_id in
(SELECT cust_id
FROM customers
WHERE cust_city = 'Asten')
AND s.prod_id in
(SELECT prod_id
FROM products
WHERE prod_id > 40000);
```

Star Transformation Example (continued)

Comparing this statement with the original one, you see that:

- Four subqueries are added
- The CUSTOMERS table disappeared in the FROM clause (line 2), together with its corresponding join predicate (line 6 on the previous page).

Note that you do not need the CUSTOMERS table in the main query, because the SELECT clause does not contain any CUSTOMERS column expression.

The four subqueries will be executed first, generating lists of values. You can use each of these lists to retrieve bitmaps. These bitmaps can be merged, resulting in one bitmap for those rows in the SALES table that match the condition in the subquery WHERE clause.

The resulting four bitmaps (one for each subquery) can be ANDed, resulting in one bitmap for those rows in the SALES table that match all four conditions. Note that this final bitmap can be very selective on the facts table without the need to perform any join operation yet. This means that the join is now easier to perform and can be driven from the facts table, without the need to produce a Cartesian product of the dimension tables, which is the basis of star join optimization, as discussed in the “Sorting and Joining” lesson.

Instructor Note

Note that the join operation itself is not optimized yet, but the join is reduced to three tables, and the final bitmap reduces the cardinality of the facts table. Conversely, three additional tables must be accessed for the subqueries.
Function-Based Indexes

You can use Oracle9i to create indexes based on column expressions (virtual columns). Function-based indexes provide an efficient mechanism for evaluating statements that contain expressions in their WHERE clauses. For example, you can use function-based indexes to create case-insensitive indexes, as the example in the slide shows.

You can create a function-based index to materialize computational-intensive expressions in the index, so that the Oracle server does not need to compute the value of the expression when processing SQL statements.

Function-based index expressions can use any function that is deterministic; that is, the returned value must not change for a given set of input values. PL/SQL functions used in defining function-based indexes must be DETERMINISTIC. The index owner needs the EXECUTE privilege on the defining function. If the EXECUTE privilege is revoked, then the function-based index is marked DISABLED. A function-based index can be created as a bitmap index.

Enabling Function-Based Indexes

Use the following command to enable function-based indexes:

```
SQL> alter session set query_rewrite_enabled = true;
```

Query rewrites are discussed in more detail in the “Materialized Views and Temporary Tables” lesson.

Instructor Note

The Oracle server does not need to compute the value of the expression when processing SELECT and DELETE statements. For INSERT and UPDATE statements the Oracle server must evaluate the expression to update the index. Demonstration script: demo11_03.sql.
Function-Based Indexes: Usage

- Materialize computational-intensive expressions
- Facilitate case-insensitive searches
- Provide simple form of data compression
- Can be used for an NLS sort index

Data Dictionary Information

You can query the data dictionary to see specific information about function-based indexes:

```sql
SQL> select table_name, index_name, index_type
2  from user_indexes
3  where index_name = 'FBI_UPPER_LASTNAME';
```

<table>
<thead>
<tr>
<th>TABLE_NAME</th>
<th>INDEX_NAME</th>
<th>INDEX_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMERS</td>
<td>FBI_UPPER_LASTNAME</td>
<td>FUNCTION-BASED NORMAL</td>
</tr>
</tbody>
</table>

```sql
SQL> select column_expression
2  from user_ind_expressions
3  where index_name = 'FBI_UPPER_LASTNAME';
```

<table>
<thead>
<tr>
<th>COLUMN_EXPRESSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPER(&quot;CUST_LAST_NAME&quot;)</td>
</tr>
</tbody>
</table>

Instructor Note

Demonstration script: demo11_04.sql
Data Dictionary Information

The result of the above query is:

<table>
<thead>
<tr>
<th>INDEX_NAME</th>
<th>INDEX_TYPE</th>
<th>COLUMN_NAME</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>SALES_PROD_BIX</td>
<td>BITMAP</td>
<td>PROD_ID</td>
<td>N/A</td>
</tr>
<tr>
<td>SALES_CUST_BIX</td>
<td>BITMAP</td>
<td>CUST_ID</td>
<td>N/A</td>
</tr>
<tr>
<td>SALES_TIME_BIX</td>
<td>BITMAP</td>
<td>TIME_ID</td>
<td>N/A</td>
</tr>
<tr>
<td>SALES_CHANNEL_BIX</td>
<td>BITMAP</td>
<td>CHANNEL_ID</td>
<td>N/A</td>
</tr>
<tr>
<td>SALES_PROMO_BIX</td>
<td>BITMAP</td>
<td>PROMO_ID</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Instructor Note**

Demonstration script: demo11_05.sql

Purpose: To demonstrate the code shown.

**Note:** The results shown from running the script table are based on the COPY_SALES table.
Summary

In this lesson, you should have learned about:

- Bitmapped indexes
- Indexes and row-access methods
- Index hints
- Star transformation
- Function-based indexes

Summary

This lesson introduced you to bitmapped indexes, star transformations, and function-based indexes.

- Bitmap indexing provides both substantial performance improvements and space savings over usual (B*-tree) indexes when there are few distinct values in the indexed columns.

- The star transformation is a cost-based query transformation aimed at executing star queries efficiently. Star transformations are used in data warehousing. One type of data warehouse design centers around what is known as a star schema, which is characterized by one or more large fact tables that contain the primary information in the data warehouse and a number of much smaller dimension tables (or lookup tables), each of which contains information about the entries for a particular attribute in the fact table.

- With function-based indexes you can create indexes based on column expressions (virtual columns). Function-based indexes provide an efficient mechanism for evaluating statements that contain expressions in their WHERE clauses, or for supporting linguistic sorting.

You can use the data dictionary views to retrieve information about your indexes.
<table>
<thead>
<tr>
<th>Schedule:</th>
<th>Timing</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>60 minutes</td>
<td>Lecture</td>
</tr>
<tr>
<td></td>
<td>60 minutes</td>
<td>Total</td>
</tr>
</tbody>
</table>
Objectives

After completing this lesson, you should be able to do the following:

• Identify the purpose and benefits of materialized views
• Create materialized views
• Enable query rewrites
• Create dimensions
• Identify the benefits of temporary tables
Materialized Views

A materialized view:
- Is an “instantiation” of a SQL statement
- Has its own data segment and offers:
  - Space management options
  - Use of its own indexes
- Is useful for:
  - Expensive and complex joins
  - Summary and aggregate data

Materialized Views

A materialized view stores both the definition of a view plus the rows resulting from the execution of the view. Like a view, it uses a query as the basis, but the query is executed at the time the view is created, and the results are stored in a table. You can define the table with the same storage parameters as any other table and place it in the tablespace of your choice. You can also index the materialized view table like other tables to improve the performance of queries executed against them.

When a query can be satisfied with data in a materialized view, the Oracle9i Server transforms the query to reference the view rather than the base tables. By using a materialized view, expensive operations such as joins and aggregations do not need to be re-executed.

Instructor Note

Materialized views (MVs) are replacing read-only snapshots. However, MVs provide greater functionality, such as the ability to allow query rewrites directly or through dimensions, as covered in this lesson. The SNAPSHOT keyword is supported in place of MATERIALIZED VIEW for backward compatibility.

There were several issues and restrictions with MVs and query rewrites in 8.1.5. For example, as soon as you have a non-join predicate in the MV definition, only exact matches will result in a query rewrite. Make sure to test any MV demonstration, and don’t improvise.
Create Materialized Views

If a query involving summaries, large or multiple joins, or even both, is likely to be used multiple times, it can be more efficient to create a materialized view of the query results. This requires a single execution of the query and the storage space to preserve the results.

If the queries are likely to be reused over time, you may also need a mechanism to update the materialized view as the base tables change. The performance and storage costs of maintaining the materialized view must be compared to the costs of re-executing the original query whenever it is needed. This is discussed in more detail later in this lesson.

The process of modifying a query to use the view rather than the base table is called a query rewrite. This is also discussed in more detail later in this lesson.

Note: This is just an example of the CREATE MATERIALIZED VIEW command; more details about optional command clauses will be discussed later during this lesson.

Instructor Note

Demonstration script: demo12_01.sql.

Purpose: The script creates the MV with the ENABLE QUERY REWRITE clause; this can also be done later with the ALTER MATERIALIZED VIEW command. The script also creates some indexes on the MV and analyzes the MV to collect statistics.
Refresh Materialized Views

- **Refresh types:**
  - COMPLETE
  - FAST
  - FORCE
  - NEVER

- **Create materialized view logs for FAST refreshes**

```sql
SQL> CREATE MATERIALIZED VIEW LOG ON ...
```

**Refresh Materialized Views**

Depending on the activity on the base tables and the accuracy of the information required, refreshing of materialized views can be done accordingly. The refresh mechanism is similar to that used by snapshots: COMPLETE, FAST, FORCE, and NEVER.

A COMPLETE refresh involves truncating existing data and reinserting all the data by re-executing the materialized view query definition.

FAST refreshes apply only the changes made since the last refresh. Two types are available:

- Using materialized view logs: In this case, all changes to the base tables are captured in a log and then applied to the materialized view.
- Using ROWID ranges: A materialized view can be refreshed fast after direct path loads, based on the ROWIDs of the new rows. Direct loader logs are required for this refresh type.

Not all materialized views can use a FAST refresh; the *Oracle9i SQL Reference* manual contains descriptions of the various view types and their refresh options.

A materialized view defined with a refresh type of FORCE refreshes with the fast mechanism if possible, or else uses a complete refresh. Force is the default refresh type.

The NEVER option suppresses all refreshes of the materialized view.

**Note:** Materialized view logs are created using the `CREATE MATERIALIZED VIEW LOG` command.
Refresh Materialized Views

- **Manual refresh**
  - By using the `DBMS_MVIEW` package

- **Automatic refresh**
  - Synchronous: Upon commit of changes made to the underlying tables—but independent of the committing transaction
  - Asynchronous: Define a refresh interval for the materialized view

Refresh Materialized Views (continued)

As well as choosing the refresh type, you can decide what mode to use to keep materialized views synchronized with their base tables.

**Manual Refresh**

Manual refreshes are performed using the `DBMS_MVIEW` package. The `DBMS_MVIEW` package provides a number of procedures and functions to manage materialized views, including the `REFRESH`, `REFRESH_DEPENDENT`, and `REFRESH_ALL_MVIEWS` procedures.

**Automatic Refresh**

Automatic refresh can be performed:

- **ON COMMIT**: When this option is specified for a materialized view, it is updated whenever changes to one of the base tables are committed.
  
  **Note**: The refresh of the MV is done as part of the transaction so the commit will take longer.

- **At a specified time**: Refresh on a materialized view can be scheduled to occur at a specified time. For example, it can be refreshed every Monday at 9:00 a.m. by using the `START WITH` and `NEXT` clauses. In order for such refreshes to occur, the instance must initiate job processes with the `JOB_QUEUE_PROCESSES` parameter.

As with refresh types, refresh modes are not all available to every materialized view. The *Oracle9i Supplied PL/SQL Packages and Types Reference* manual describes the modes and when each can be used.

Refer to the *Oracle9i Data Warehousing Guide* for more information on Materialized views.
Materialized Views: Manual Refresh

- **Specific materialized views**
  
  ```sql
  DBMS_MVIEW.REFRESH
  ('F WEEK_PSCAT_COSTS_MV', PARALLELISM => 10);
  ```

- **Materialized views based on one or more tables**
  
  ```sql
  DBMS_MVIEW.REFRESH_DEPENDENT
  ('CAL_MONTH_SALES_MV');
  ```

- **All materialized views due for refresh**
  
  ```sql
  DBMS_MVIEW.REFRESH_ALL_MVIEWS(n);
  ```
  - Returns a binary integer value as an output parameter

---

**Manual Refresh (Continued)**

The following is a list of possible refresh scenarios for materialized views:

- Refresh specific materialized views using the `REFRESH` procedure
- Refresh all materialized views that depend on a given set of base tables using the `REFRESH_DEPENDENT` procedure
- Refresh all materialized views that have not been refreshed since the last bulk load to one or more detail tables using the `REFRESH_ALL_MVIEWS` procedure

The procedures in the package use a number of parameters to specify the following:

- Refresh method
- Whether to proceed if an error is encountered
- Whether to use a single transaction (consistent refresh)
- The rollback segment to use

You need server job queues to run the refresh job. Therefore the appropriate initialization parameters, `JOB_QUEUE_PROCESSES` and `JOB_QUEUE_INTERVAL`, must be set to enable job queue refresh processes.
Query Rewrites

- To use a materialized view instead of the base tables, a query must be rewritten.
- Query rewrites are transparent to applications.
- Query rewrites do not require special privileges on the materialized view.
- A materialized view can be enabled or disabled for query rewrites.

Query Rewrites

Because accessing a materialized view may be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications. In this respect, their use is similar to the use of an index.

Users do not need explicit privileges on materialized views to use them. Queries executed by any user with privileges on the underlying tables can be rewritten to access the materialized view.

A materialized view can be enabled or disabled. A materialized view that is enabled is available for query rewrites.
Query Rewrites

The optimizer rewrites queries in order to use materialized views:

- The `QUERY REWRITE` privilege allows users to enable materialized views.
- The `DBMS_OLAP` package has options to use materialized views.

Query Rewrites (continued)

To enable or disable individual materialized views for query rewrites, you must have the `GLOBAL QUERY REWRITE` or the `QUERY REWRITE` system privilege. Both versions of the privilege allow you to enable materialized views in your own schema. The `GLOBAL` version allows you to enable any materialized views you own, whereas the simple `QUERY REWRITE` privilege requires that the base tables, as well as the views, be in your own schema.

The `DBMS_OLAP` supplied package provides a collection of materialized view analysis and advisory functions. For more details, see the Oracle9i Supplied PL/SQL Packages and Types Reference.
Query Rewrites

- Use EXPLAIN PLAN or AUTOTRACE to verify that query rewrites occur.
- Check the query response:
  - Fewer blocks are accessed
  - Response time should be significantly better

Query Rewrites (continued)

The best method to detect whether query rewrites occur is to use the EXPLAIN PLAN command or the AUTOTRACE setting in SQL*Plus. You should also notice improved response time if a materialized view is used by the optimizer.

Note: There are several system privileges that control whether you are allowed to create materialized views and modify them, and whether query rewrites are enabled for you. Contact your local database administrator to set this up properly, or refer to the Oracle9i Administrator’s Guide for details.

There are also many data dictionary views that contain information about materialized views; check the Oracle9i Performance Guide and Reference manual and the Oracle9i Reference for details.

Instructor Note

Note that the main audience for this class is developers, not database administrators; try to avoid DBA-related topics. Also note that there are some bugs with SQL*Plus AUTOTRACE not showing the query rewrite, because it uses an implicit second database session that does not inherit all settings from your main session. Using EXPLAIN is safer.

Instructor Note (for page 12-12)

Demonstration script: demo12_02.sql
Create Materialized Views: Syntax Options

```
CREATE MATERIALIZED VIEW mview_name
   [TABLESPACE ts_name]
   [PARALLEL (DEGREE n)]
   [BUILD {IMMEDIATE | DEFERRED}]
   [REFRESH {FAST | COMPLETE | FORCE
       | NEVER | ON COMMIT}]
   [{ENABLE | DISABLE} QUERY REWRITE]

AS SELECT ... FROM ...
```

Create Materialized Views: Syntax Options

The `CREATE MATERIALIZED VIEW` syntax is similar to the `CREATE SNAPSHOT` command, which it replaces. There are some additional options:

- The `BUILD IMMEDIATE` option causes the materialized view to be populated when the `CREATE` command is executed. This is the default behavior. You can choose the `BUILD DEFERRED` option, which creates the structure but does not populate it until the first refresh occurs.

- Instead of the `BUILD` option, you can also specify `ON PREBUILT TABLE` when you want an existing summary table to be the source of a materialized view.

- The `ENABLE/DISABLE QUERY REWRITE` clause determines whether query rewrites are automatically enabled for the materialized view.

**Note:** The default options in the syntax are underlined. This is not the complete syntax; refer to the *Oracle9i SQL Reference* for more details.

**Instructor Note**
You can use `ON PREBUILT TABLE` to convert an existing table to a preinitialized materialized view. There is an additional clause for this option, `WITH/WITHOUT REDUCED PRECISION`. This clause, which defaults to `WITHOUT REDUCED PRECISION`, determines whether or not the columns in the subquery must have the same precision as the corresponding columns in the table.
Enabling and Controlling Query Rewrites

- Query rewrites are available only with cost-based optimization.

```
QUERY_REWRITE_ENABLED = {true|false}
QUERY_REWRITE_INTEGRITY = {enforced|trusted|stale_tolerated}
```

- Optimizer hints to influence query rewrites are REWRITE and NOREWRITE.

Enabling and Controlling Query Rewrites

OPTIMIZER_MODE: Query rewrites are available only under cost-based optimization.

QUERY_REWRITE_ENABLED: This parameter can be set to FALSE to suppress query rewrites.
This is a dynamic instance/session parameter.

QUERY_REWRITE_INTEGRITY: This is also a dynamic instance/session parameter.
It accepts the following values:

- ENFORCED (the default) enables query rewrites only if the optimizer can guarantee consistency. Only updated materialized views and enabled validated constraints are used for query rewrites.

- TRUSTED allows query rewrites based on declared (not necessarily enforced) relationships. All updated materialized views and dimensions (discussed later in this lesson) and constraints with the RELY flag are used for query rewrites.

- STALE_TOLERATED allows query rewrites to use materialized views that have not been refreshed since the last DML operation and relationships that have been declared.

Note: There are no object privileges associated with query rewrites. Users with access to the base tables implicitly benefit from query rewrites.
You can use the hint REWRITE to restrict the materialized views that are considered for query rewrites. The NOREWRITE hint is available to suppress query rewrites.
Query Rewrite Example

The execution plan shows that the materialized view is accessed instead of joining all four base tables to produce the result.

A REWRITE/NOREWRITE hint overrides a materialized view’s definition, set in the CREATE or ALTER MATERIALIZED VIEW command with the ENABLE QUERY REWRITE clause.

This example shows a transparent query rewrite where the query exactly matches the materialized view definition. The next page shows an example of a query that does not match the materialized view definition.

Instructor Note

Tell the students to analyze the materialized view, as one of the previous demonstrations did.

Demonstration script: demo12_03.sql.
Query Rewrite Example

\begin{verbatim}
SQL> SELECT t.week_ending_day, t.calendar_year, p.prod_subcategory, 
    4 sum(c.unit_cost) AS dollars 
FROM costs c, times t, products p 
WHERE c.time_id = t.time_id 
AND c.prod_id = p.prod_id 
AND t.calendar_year = '1999' 
GROUP BY t.week_ending_day, t.calendar_year, 
    p.prod_subcategory 
HAVING sum(c.unit_cost) > 10000;
\end{verbatim}

\begin{verbatim}
SQL> SELECT week_ending_day, prod_subcategory, dollars 
from fweek_pscat_costs_mv 
where calendar_year = '1999' 
and dollars > 10000;
\end{verbatim}

Query Rewrite Example (continued)

The optimizer can use the materialized view created earlier to satisfy the query.

Note: This query does not exactly match the materialized view definition. You have added a nonjoin predicate on line 8 and a \texttt{HAVING} clause on the last line. The nonjoin predicate is merged into the rewritten query against the materialized view and the \texttt{HAVING} clause is translated into a second component of the \texttt{WHERE} clause.

Instructor Note

Demonstration script: demo12_04.sql
Dimensions: Overview

- Dimensions are data dictionary structures that define hierarchies based on existing columns.
- Dimensions are optional but recommended because they:
  - Enable additional query rewrites
  - Help document hierarchies
  - Can be used by OLAP tools

Dimensions

Dimensions are data dictionary structures that define hierarchies based on columns in existing database tables. Although they are optional, they are recommended because they:

- Enable additional rewrite possibilities without the use of constraints (Implementation of constraints may not be desirable in a data warehouse for performance reasons).
- Help document dimensions and hierarchies explicitly.
- Can be used by OLAP tools.

A dimension is an object stored in the data dictionary that enables a query to leverage the results of an existing summary to produce another summary much cheaper. For example if the original query had 1700 I/Os, using the materialized view would take about 170 I/Os and the query on using a dimension would take approximately 17 I/Os.

Instructor Note

If you are running out of time, you can say that dimensions enable additional query rewrite capabilities, specifically targeted at data warehouse environments; then you can refer them to the excellent chapters in the Oracle9i Data Warehousing Guide manual for more details. You can then continue to discuss temporary tables, the last topic of this lesson.
Dimensions and Hierarchies

Dimensions describe business entities such as products, departments, and time in a hierarchical, categorized manner. A dimension can consist of one or more hierarchies. In the example shown in the slide, the time dimension consists of three hierarchies.

Each hierarchy comprises multiple levels. Each value at a lower level in the hierarchy is the child of one and only one parent at the next higher level. A hierarchy consists of 1-to-n relationships between levels, with the parent level representing a level of aggregation of the child level. In the example, the Fiscal Time hierarchy consists of the fiscal week, fiscal month, fiscal quarter, and fiscal year.

A level in the hierarchy has a level key that can identify one or more dimensional attributes. In the example, month is a level where the level key MONTH identifies the MONTH_NAME attribute.

You can use level keys and their related attribute names interchangeably in a query.
Dimensions: Example Table

You have a TIMES table containing time information in several separate columns. On the next two pages you will create a dimension, TIMES_DIM, on this table.
Dimensions and Hierarchies

Dimensions can be based on columns in a single table, as shown in this first example. Later during this lesson you see that you can also base dimensions on columns from multiple tables. When creating dimensions, you must define and name hierarchy levels based on the TIMES table column names.

In this case, the TIMES_DIM dimension has four levels:

- The highest level in the hierarchy consists of the CALENDAR_YEAR column.
- The next level is derived from the CALENDAR_QUARTER_DESC column.
- The next level is derived from the CALENDAR_MONTH_DESC column.
- The lowest level is based on the TIME_ID column.

- There are three attributes. The second level has the CALENDAR_QUARTER_DESC column as the key, and DAYS_IN_CAL_QUARTER as the attribute. The third level has the CALENDAR_MONTH_DESC column as the key and CALENDAR_MONTH_NAME as the attribute. The fourth level has the TIME_ID column as the key, and DAY_NAME as the attribute.

Note that each attribute value maps to a single key value and vice versa.
Create Dimensions and Hierarchies

```
SQL> CREATE DIMENSION times_dim
2   LEVEL day     IS TIMES.TIME_ID
3   LEVEL month   IS TIMES.CALENDAR_MONTH_DESC
4   LEVEL quarter IS TIMES.CALENDAR_QUARTER_DESC
5   LEVEL year    IS TIMES.CALENDAR_YEAR
6   HIERARCHY cal_rollup    (
7       day     CHILD OF
8       month   CHILD OF
9       quarter CHILD OF
10       year
11     )
12   ATTRIBUTE day DETERMINES (day_name)
13   ATTRIBUTE month DETERMINES (calendar_month_name)
14   ATTRIBUTE quarter DETERMINES
15                       (days_in_cal_quarter);
```

Defining Dimensions and Hierarchies

Hierarchical relationships can be declared between two columns in a dimension table, or between two columns in different tables in the case of normalized or snowflake schemas to support such rewrites. A new schema object type, dimension, is provided for this purpose.

The parent-child relationship along a dimension is stronger than referential integrity alone because it requires that the child join key be mandatory (that is, defined as `NOT NULL`).

The creator of a dimension must ensure (optionally, with database constraints in the case of a normalized dimension) that the columns of each hierarchy level are non-NULL and that hierarchical integrity is maintained.

Note that the key and attributes for a given level must be based on columns from the same table.

The example shows a single hierarchy within the time dimension, but it is possible to have multiple hierarchies. For example, another hierarchy can be created to link sales date with a fiscal calendar or season.

The system privilege, `CREATE DIMENSION`, is required to create a dimension in your own schema based on tables that are within the same schema. Another privilege, `CREATE ANY DIMENSION`, enables you to create dimensions in any schema.

In the example shown, the `TIMES_DIM` dimension is based on the `TIMES` table.
Dimensions Based on Multiple Tables

- **Example:** A GEOGRAPHY dimension is based on CITY, STATE, and REGION tables.
- Define foreign key constraints on the child tables to maintain dimension validity.
- All attributes and keys for one level must belong to the same table.

Dimensions Based on Multiple Tables

A dimension can use columns from several tables. For example, a GEOGRAPHY dimension can use the following tables and columns:

- **CITY** table containing CITY_NAME, OFFICE_ADDRESS, POPULATION, and STATE_CODE
- **STATE** table consisting of STATE_CODE, STATE_NAME, and REGION_ID
- **REGION** table comprising REGION_ID, REGION_NAME, and COUNTRY

In this case, foreign key constraints should be defined on a child table to reference the primary key of the parent table. An example may be CITY.STATE_CODE referencing STATE.STATE_CODE. This helps maintain dimension validity.

Note that all columns for a given level are stored in the same table.
Dimensions with Multiple Hierarchies

The pair of hierarchies shown in the slide can be created within a single dimension. Following is the statement to do this:

```
SQL> CREATE DIMENSION times_dim
2     LEVEL day         IS TIMES.TIME_ID
3     LEVEL month       IS TIMES.CALENDAR_MONTH_DESC
4     LEVEL quarter     IS TIMES.CALENDAR_QUARTER_DESC
5     LEVEL year        IS TIMES.CALENDAR_YEAR
6     LEVEL fis_week    IS TIMES.WEEK_ENDING_DAY
7     LEVEL fis_month   IS TIMES.FISCAL_MONTH_DESC
8     LEVEL fis_quarter IS TIMES.FISCAL_QUARTER_DESC
9     LEVEL fis_year    IS TIMES.FISCAL_YEAR
10    HIERARCHY cal_rollup    ( 
11       day     CHILD OF
12       month   CHILD OF
13       quarter CHILD OF year )
14    HIERARCHY fis_rollup    ( 
15       day     CHILD OF
16       fis_week CHILD OF
17       fis_month CHILD OF
18       fis_quarter CHILD OF fis_year );
```
Temporary Tables

- Temporary tables retain data for the duration of a transaction or session.
- Definitions persist in the data dictionary.
- Data is only visible to the session.
- Temporary tables:
  - Use sort space in memory
  - Allocate temporary extents, if needed
Creating Temporary Tables

You can use temporary tables to improve performance when you run complex queries. Running multiple such queries is relatively slow because the tables are accessed multiple times for each returned row. It is faster to cache the values from a complex query in a temporary table, then run the queries against the temporary table.

For example, even with a view defined to simplify further queries, as shown in the slide, the queries against the view may be slow because the contents of the view are recalculated each time.

You can use a temporary table to run the computation once, and cache the result in later SQL queries and joins.

```sql
SQL> CREATE OR REPLACE VIEW sales_detail
2  AS
3  SELECT cu.cust_last_name, cu.cust_email,
4       , cu.cust_income_level
5       , pr.prod_name, ch.channel_desc
6       , pm.promo_name, sa.amount_sold
7  FROM customers cu, products pr,
8       , channels ch, promotions pm, sales sa
9  WHERE sa.cust_id = cu.cust_id
10  AND sa.prod_id = pr.prod_id
11  AND sa.channel_id = ch.channel_id
12  AND sa.promo_id = pm.promo_id
13  AND sa.time_id between
14       '01-DEC-1999' and '31-DEC-1999'
View created.
```
Creating Temporary Tables

```
SQL> CREATE GLOBAL TEMPORARY TABLE sales_detail_temp
2  ( cust_last_name     VARCHAR2(50)
3  , cust_income_level  VARCHAR2(30)
4  , cust_email         VARCHAR2(30)
5  , prod_name          VARCHAR2(50)
6  , channel_desc       VARCHAR2(20)
7  , promo_name         VARCHAR2(20)
8  , amount_sold        NUMBER)
9  ON COMMIT PRESERVE ROWS;

Table created.
```

```
SQL> INSERT INTO sales_detail_temp
2  SELECT * FROM sales_detail;
37830 rows created.
```

Creating Temporary Tables (Continued)

The example in the slide shows the command to create a temporary table where rows are visible only within the session. The temporary table is then populated with rows from the previously defined view. The computation is performed once and the cached results are available for future queries.

The clauses that control the duration of the rows are:

- **ON COMMIT DELETE ROWS** to specify that rows are visible only within the transaction (This is the default behavior).
- **ON COMMIT PRESERVE ROWS** to specify that rows are visible for the entire session.

Although the command does not create extents for the table, a session can query the table without receiving an error.

**Instructor Note**

There is a bug in 8.1.5 that shows the **DURATION** to be **SYS$TRANSACTION** even if the table is defined with **ON COMMIT PRESERVE ROWS**. The data in the table, however, is visible throughout the session. Demonstration script: demo12_05.sql.
Creating Temporary Tables

The view DBA_TABLES contains columns to indicate that the table is a temporary table. The column DURATION shows whether data in the table persists for a whole session or only within the transaction.
Summary

In this lesson, you should have learned how to:

- Create materialized views
- Enable query rewrites using materialized views
- Enable additional query rewrite capabilities with dimensions and hierarchies
- Create temporary tables to hold data within the scope of a session or transaction

This lesson introduced you to materialized views, query rewrites, dimensions and hierarchies, and temporary tables:

- A materialized view stores both the definition of a view plus the rows resulting from the execution of the view. Like a view, it uses a query as the basis, but the query is executed at the time the view is created and the results are stored in a table.

- Because accessing a materialized view can be significantly faster than accessing the underlying base tables, the optimizer rewrites a query to access the view when the query allows it. The query rewrite activity is transparent to applications.

- Dimensions are data dictionary structures that define hierarchies based on columns in existing database tables. Dimensions describe business entities such as products, departments, and time in a hierarchical, categorized manner. A dimension can consist of one or more hierarchies. Each hierarchy comprises multiple levels.

- The Oracle9i Server enables you to create temporary tables. Temporary tables can improve performance significantly by holding temporary data for reuse within your transaction or session.
## Alternative Storage Techniques

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Timing</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>45 minutes</td>
<td>Lecture</td>
</tr>
<tr>
<td></td>
<td>45 minutes</td>
<td>Total</td>
</tr>
</tbody>
</table>
Objectives

After completing this lesson, you should be able to identify and describe the benefits of:

• Index-organized tables
• External tables
Storing User Data

The Oracle9i Server offers a wide range of techniques to store user data: regular tables, index-organized tables, partitioned tables, and external tables.

Partitioning is not covered in this course. This lesson concentrates on the other, nonregular table storage techniques: index-organized tables and external tables.

Changing the physical storage characteristics of tables sometimes results in significant performance improvements for certain SQL statements. However, be aware that there are usually negative side effects.

Data maintenance is more complicated for nonregular tables.

Change regular tables into index-organized tables or external tables when the advantages outweigh the disadvantages significantly and when these disadvantages do not violate any minimum performance constraints.

Instructor Note

Partitioning is covered in Oracle9i: Implementing Scalable Systems.
Index-Organized Tables

Indexed access on table

Accessing index-organized table

ROWID

Non-key columns
Key column
Row header

Index-Organized Tables: Overview

An indexed-organized table (IOT) is like a regular table with an index on one or more of its columns, but instead of maintaining two separate segments for the table and the B*-tree index, the Oracle server maintains one single B*-tree structure which contains:

- The primary key value
- Other (non-key) column values for the row

The B*-tree structure, which is based on the primary key of the table, is organized like an index. The leaf blocks in this structure contain the rows instead of the ROWIDs. This means that the rows in the index-organized table are always maintained in the order of the primary key.

Because index-organized tables do not have real ROWIDs, the concept of logical ROWIDs is introduced in Oracle8i. This enables you to create additional indexes on index-organized tables; this was not possible in Oracle8 release 8.0.

Because large rows of an index-organized table can destroy the dense and efficient storage of the B*-tree structure, in Oracle8i you can store part of the row in another segment, called an overflow area. This is discussed on the following pages.
IOT Performance Characteristics

- B*-tree stores full rows
- Sorted rows
- Logical ROWIDs
- Fast, key-based access to table data

You can access an index-organized table by using either the primary key or a combination of columns that constitute the leading part of the primary key.

IOTs provide fast, key-based access for queries involving exact match (equality operator) or range searches on the primary key.

Before Oracle8i, any other access to rows in an index-organized table required a full scan of the B*-tree structure. That is why Oracle8 introduced the concept of logical ROWIDs, which you can use to create additional indexes on an index-organized table. Logical ROWIDs internally have the UROWID data type.

Because the rows are ordered in the IOT, full scans on an index-organized table return rows in a primary key sequence.

Because there is no duplication of primary key values (compared with regular tables: index segment and data segment), IOTs use less storage. Index organization is useful for a table that is frequently accessed using the primary key and has only a few, relatively short nonkey columns.

Instructor Note

The statement in the fourth paragraph about returning the rows in primary key sequence is true only for serial execution; it is no longer true with parallel execution. Also, the first bullet in the slide is not always true: the concept of row overflow is introduced on the following pages.

Oracle9i: SQL Tuning Workshop 13-5
IOT Limitations

- Must have a primary key
- Cannot be part of an index cluster or hash cluster
- Cannot contain LONG columns (although LOB columns are allowed)

Index-organized tables must have a primary key. This is the unique identifier and is used as the basis for ordering; there is no ROWID to act as a unique identifier in the B*-tree structure.

Index-organized tables cannot be part of an index cluster or a hash cluster.

Index-organized tables cannot include LONG columns, but they can contain LOB columns. Because index-organized tables are B*-tree structures, they are subject to fragmentation as a result of incremental updating. You can use the ALTER TABLE ... MOVE command to rebuild the index-organized table:

```
ALTER TABLE iot_tablename MOVE [OVERFLOW...];
```

Specifying the optional OVERFLOW clause causes the overflow segment to be rebuilt as well. Overflow segments are explained on the following pages.

Instructor Note

Oracle8i removed the IOT restrictions regarding secondary indexes and additional UNIQUE constraints. You can mention the absence of real ROWIDs as an IOT limitation, although the implementation of logical ROWIDs (with the UROWID data type) resolved all related shortcomings.

Note that the MOVE option for the ALTER TABLE statement was new in Oracle8i.
When to Use Index-Organized Tables

Applications that use complex data objects have many different indexing requirements. For example, information-retrieval applications may require a keyword search or a search for similar words.

Index-organized tables are particularly useful for applications accessing and manipulating complex data. Examples are information-retrieval, spatial, and OLAP applications.

No special considerations exist for using most SQL statements against an index-organized table. The structure of the table should be completely transparent to the user.

The Oracle utilities also support index-organized tables. SQL*Loader, using direct path mode, can load data directly into an index-organized table. This kind of loading can be quicker than loading into a standard table and then building the indexes afterwards.

Note: IOTs are supported by Oracle8i and later utilities.
Creating Index-Organized Tables

The ORGANIZATION INDEX clause of the CREATE TABLE statement specifies that you create an index-organized table. You must specify a primary key constraint when creating index-organized tables. If you try to create an index-organized table without a primary key, the following error is generated:

ORA-25175: no PRIMARY KEY constraint found

PCTTHRESHOLD specifies the percentage of space reserved in a block for a single row in an index-organized table. If a row exceeds the size calculated based on this value, all columns after the column named in the INCLUDING clause are moved to the overflow segment. If OVERFLOW is not specified, then rows exceeding the threshold are rejected. PCTTHRESHOLD defaults to 50 and must be a value from 0 to 50.

INCLUDING column_name specifies a column at which to divide an index-organized table row into index and overflow portions. All columns that follow column_name are stored in the overflow data segment. If this is not specified and a row size exceeds PCTTHRESHOLD, all columns except the primary key columns are moved to the overflow area. The column is either the name of the last primary key column or any nonprimary key column.

OVERFLOW specifies that index-organized table rows exceeding the specified threshold be placed in the data segment defined by segment_attr_clause, which specifies the tablespace, storage, and block utilization parameters.

Oracle9i: SQL Tuning Workshop 13-8
IOT Row Overflow

Large rows in an index-organized table can destroy the dense B*-tree storage. You can overcome this problem by using an overflow area.

Note that you need additional I/Os to retrieve these large rows because the rows are stored in two pieces. This results in a decrease in performance compared to an IOT with shorter records that are mostly stored entirely in the IOT segment alone.

When an index-organized table is created by specifying an OVERFLOW clause, the following three segments are created in the database:

- A logical table with the name defined in the CREATE TABLE clause
- An index with the same name as the primary key constraint
- A table to accommodate the overflow row pieces (The name of this table is SYS_IOT_OVER_n, where n is the OBJECT_ID of the index-organized table as seen from USER_OBJECTS.)

Note: If you create an index-organized table without specifying an OVERFLOW clause, only the first two segments are created. Give the primary key constraint a name, so that the index segment receives a meaningful name (as opposed to a system-generated one.)
Retrieving IOT Information from the Data Dictionary

Use the following query to list the index-organized tables and information related to their structure:

```
SQL> SELECT t.table_name as "IOT",
          o.table_name as "Overflow",
          i.index_name as "Index",
          o.tablespace_name as "Overflow TS",
          i.tablespace_name as "Index TS",
          i.pct_threshold as "Threshold"
FROM   user_tables  t
       ,      user_tables  o
       ,      user_indexes i
WHERE  t.table_name = o.iot_name
       AND    t.table_name = i.table_name;
```
External Tables

External Tables: Overview

Oracle9i provides a way to access data in external sources as if it were in a table in the database. Oracle allows you read-only access to data in external tables. External tables are defined as tables that do not reside in the database and can be in any format for which an access driver is provided. By providing Oracle with metadata describing an external table, Oracle is able to expose the data in the external table as if it were data residing in a regular database table. The external data can be queried directly and in parallel using SQL.

You can select, join, or sort external table data. You can also create views and synonyms for external tables. However, no DML operations (UPDATE, INSERT, or DELETE) are possible, and no indexes can be created on external tables.
External Tables Performance Characteristics

- External tables are read-only tables in which the data is stored outside the database in flat files.
- The metadata for an external table is created using a `CREATE TABLE` statement.
- With the help of external tables, Oracle data can be stored outside the database as flat files.
- Specifying a `PARALLEL` clause is of value only when dealing with large amounts of data. Otherwise, it is not advisable to specify a `PARALLEL` clause, and doing so can be detrimental.

External Tables Performance Characteristics

When you access the external table through a SQL statement, the fields of the external table can be used just like any other field in a “normal” table. In particular, you can use the fields as arguments for any SQL built-in function, PL/SQL function, or Java function. Thus you can manipulate data from the external source. For data warehousing, you can do more sophisticated transformations in this way than you can with simple data type conversions. You can also use this mechanism in data warehousing to do data cleansing.

Parallel Access to External Tables

After the metadata for an external table is created, you can query the external data directly and in parallel using SQL. As a result, the external table acts as a view, which you use to run any SQL query against external data without loading the external data into the database. The degree of parallel access to an external table is specified using standard parallel hints and with the parallel clause. Using parallelism on an external table allows for concurrent access to the data files that comprise an external table. Whether a single file is accessed concurrently or not is dependent upon the access driver implementation and attributes of the data file(s) being accessed (for example, record formats or seekable media).
External Tables Limitations

- The data can be queried using SQL but you cannot use DML.
- Indexes cannot be created on external tables.
- The `ANALYZE` statement is not supported for gathering statistics for external tables. The `DBMS_STATS` package should be used for gathering statistics for external tables.

External Table Limitations

- Whereas you can use external tables to access data stored in a file outside of the database, you can not perform DML on that external data.
- Indexes are not supported on external tables.
- To gather statistics for an external table, use the `DBMS_STATS` package. The `ANALYZE` command is not supported.
Why Use External Tables?

- Provides a valuable means for performing basic extraction, transformation, and transportation (ETT) tasks that are common for data warehousing
- Are transparent to users and applications
- Are a complement to the existing SQL*Loader functionality
- Are especially useful for environments where:
  - The complete external source must be joined with existing database objects and transformed in a complex manner.
  - The external data volume is large and used only once.

When to Use External Tables

External tables provides a valuable means for performing basic extraction, transformation, and transportation (ETT) tasks that are common for data warehousing. They are especially useful for environments where the complete external source must be joined with existing database objects and transformed in a complex manner, or the external data volume is large and used only once.
Creating External Tables

The ORGANIZATION EXTERNAL clause of the CREATE TABLE statement specifies that you create an external table. There are several clauses that identify where the external file is located and how the external file is structured.

A directory object is required prior to using the CREATE TABLE ... ORGANIZATION EXTERNAL statement. This directory object identifies the location of the external file on the operating system.

**Note:** For details about how to create an external table, see the Oracle9i Database Administrator’s Guide manual.

**Instructor Note**

The details of creating an external table are covered in the Introduction to Oracle9i: SQL course. That course is a prerequisite, so do not get into the specifics here.
Retrieving External Tables Information from the Data Dictionary

Use the following query to list the external tables and information related to their structure:

```sql
SQL> SELECT t.table_name
    2   , t.type_name
    3   , t.default_directory_name
    4   , t.reject_limit
    5   , t.access_type
    6   , t.access_parameters
    7   , l.location
    8   , l.directory_name
    9 FROM user_external_tables  t
   10   , user_external_locations  l
   11  WHERE t.table_name = l.table_name;
```
Summary

In this lesson, you should have learned how to:

- Create index-organized tables (IOTs)
  - Identify IOT performance characteristics
  - Identify IOT Limitations
  - Control IOT row overflow
- Create external tables
  - Identify external tables performance characteristics
  - Identify external tables limitations

Summary

This lesson introduced you to alternative storage techniques including index-organized tables, index clusters and hash clusters.

- An indexed-organized table (IOT) is like a regular table with an index on one or more of its columns, but instead of maintaining two separate segments for the table and the B*-tree index, the Oracle server maintains one single B*-tree structure which contains the primary key value and other (non-key) column values for the row.
- An external table is a data source that is located outside of the Oracle database. After the external table is set up using the CREATE TABLE ... ORGANIZATION EXTERNAL syntax, you can query this data source using a SELECT statement.
Data Warehousing Considerations

<table>
<thead>
<tr>
<th>Schedule:</th>
<th>Timing</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40 minutes</td>
<td>Lecture</td>
</tr>
<tr>
<td></td>
<td>40 minutes</td>
<td>Total</td>
</tr>
</tbody>
</table>
Objectives

After completing this lesson, you should be able to do the following:

• Create improved aggregation statements
• Simultaneously insert data into multiple tables
• Use the \texttt{MERGE} statement to conditionally update or insert data

Objectives

Data warehousing applications often have requirements to extract data from multiple sources, transform it to different formats, transport it to different platforms, and load it into database tables. The Oracle9i database has a number of new features that streamline these processes.

For more information, see \textit{Oracle9i Data Warehousing Guide} and \textit{Oracle9i SQL Reference}.
The **WITH** Clause: Overview

- Enables you to use the same query block in a `SELECT` statement when it occurs more than once within a complex query
- Retrieves the results of a query block and stores them in the user’s temporary tablespace
- Can improve performance

Using the `WITH` clause, you can reuse the same query through materialization when it is costly to evaluate the query block and it occurs more than once within a complex query.
The with Clause

• Name a query block in a select statement to reference it more than once within a query.
• Can hold multiple query blocks, separated by commas
• Is resolved as in-line views or temporary tables

The WITH Clause

Use the WITH clause to define a query block before using it in a query. This can improve performance, and it certainly improves readability and maintainability of your SQL statements. You can also use the WITH clause to isolate the business question from the data gathering.

A query name is visible to all WITH element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).

The WITH clause is internally resolved as an in-line view or a temporary table; the cost-based optimizer chooses the appropriate resolution.

Note: The new WITH clause does not always offer a significant improvement in performance. However it is an ease-of-use feature.
The WITH Clause: Example

Find all product subcategories whose average product list price is above one-half the average product list price of the whole company.

```sql
SQL> WITH
2    summary AS (
3    select prod_subcategory
4    ,      avg(prod_list_price) AS avg_subcat
5    from   products
6    group by prod_subcategory)
7 SELECT prod_subcategory, avg_subcat
8  FROM   summary
9  WHERE  avg_subcat > (     
10         SELECT AVG(avg_subcat) * 0.5
11         FROM   summary)
12* ORDER BY avg_subcat DESC;
```

The WITH Clause: Example

The output from the query identifies product subcategories as having list prices one-half above the average list price for the company.

The Autotrace output for the above query is as follows:

Statistics

- 4 recursive calls
- 22 db block gets
- 720 consistent gets
- 20 physical reads
- 520 redo size
- 2250 bytes sent via SQL*Net to client
- 634 bytes received via SQL*Net from client
- 3 SQL*Net roundtrips to/from client
- 2 sorts (memory)
- 0 sorts (disk)
- 25 rows processed

Instructor Note

Demonstration script: demo14_01.sql and demo14_02.sql

Purpose: These scripts run the statements shown on pages 5 and 6 to demonstrate the WITH clause statistics.
The WITH Clause: Implementation

Equivalent query, less powerful syntax:

```
SQL> SELECT prod_subcategory
2  ,  AVG(prod_list_price) AS avg_subcat
3  FROM   products
4  GROUP BY prod_subcategory
5  HAVING AVG(prod_list_price) >
6       (SELECT AVG(prod_list_price) * 0.5
7            FROM products)
8  ORDER BY avg(prod_list_price) DESC;
```
The WITH Clause: Usage Notes

- Use it only for SELECT statements.
- A query name is visible to all WITH element query blocks defined after it and the main query block itself.
- When the query name is the same as an existing table name, the parser searches from the inside out. The query block name takes precedence over the table name.
- The WITH clause can hold more than one query. Each query is separated by a comma.

A query name is visible to all WITH element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).

A query name can be the same as some persistent table name or query name in the WITH list of another query block. If this happens, the parser searches for the right definition inside out (in terms of query block nesting). The innermost query name definition is used for resolution. Users must make sure that it does not conflict with permanent tables if that is not their intention. This is similar to the C language, in which a variable has priority when its name is the same as some global variable.
The **WITH** Clause: Benefits

Using the **WITH** clause adds business value because:

- The business question is isolated from data gathering.
- The query is easy to read.
- A clause is evaluated only once, even if it appears multiple times in the query.
- Performance is improved because the instantiated subquery clause must be calculated only once.
Multitable INSERT Statements

- Allow the `INSERT ... SELECT` statement to insert rows into multiple tables as part of a single DML statement
- Can be used in data warehousing systems to transfer data from one or more operational data sources to a set of target tables
- Can be used for refreshing materialized views

The `INSERT ... SELECT` statement with the new syntax can be parallelized and used with the direct load mechanism. The multitable `INSERT` statement inserts computed rows derived from the rows returned from the evaluation of a subquery. There are two forms of the multitable `INSERT` statement: unconditional and conditional. For the unconditional form, an `INTO` clause list is executed once for each row returned by the subquery. For the conditional form, `INTO` clause lists are guarded by `WHEN` clauses that determine whether the corresponding `INTO` clause list is executed.

An `INTO` clause list consists of one or more `INTO` clauses. The execution of an `INTO` clause list causes the insertion of one row for each `INTO` clause in the list.

An `INTO` clause specifies the target into which a computed row is inserted. The target specified can be any table expression that is legal for an `INSERT ... SELECT` statement. However, aliases cannot be used. The same table can be specified as the target for more than one `INTO` clause.

An `INTO` clause also provides the value of the row to be inserted using a `VALUES` clause. An expression used in the `VALUES` clause can be any legal expression, but can refer only to columns returned by the select list of the subquery. If the `VALUES` clause is omitted, the select list of the subquery provides the values to be inserted. If a column list is given, each column in the list is assigned a corresponding value from the `VALUES` clause or the subquery. If no column list is given, the computed row must provide values for all columns in the target table.
Multitable INSERT Statement

The example in the slide inserts the values of order date, product_id and sum(quantity) into the product_activity table and order date, product_id and sum(unit_price) into the prod_sales table using a single statement. The values are obtained through a subquery selecting from a join between the orders and order_items tables.

This is the alternative syntax using two individual INSERT statements.

```
INSERT ALL INTO product_activity
VALUES(today, product_id, quantity)
INTO prod_sales
    (prod_id, time_id, amount_sold, quantity_sold)
VALUES(product_id, today, total, quantity)
SELECT order_date today, product_id,
    SUM(unit_price) total,
    SUM(quantity) quantity
FROM orders, order_items
WHERE orders.order_id = order_items.order_id
GROUP BY order_date, product_id;
1330 rows created.
```

```
665 rows created.
```

```
665 rows created.
```

Oracle9i: SQL Tuning Workshop 14-10
Conditional Multitable Inserts

Conditional multitable inserts use the WHEN clause to specify conditions. This can further be classified into:

- Conditional FIRST: The first true condition will be inserted.
- Conditional ALL: All true conditions will be inserted.

Conditional FIRST has the following signature:

```sql
INSERT FIRST
WHEN <condition> into <table>(col, col,...)
WHEN <condition> into <table>(col, col,...)
ELSE <table>(col, col,...)
SELECT...;
```

Conditional ALL has the following signature:

```sql
INSERT ALL
WHEN <condition> into <table>(col, col,...)
WHEN <condition> into <table>(col, col,...)
ELSE <table>(col, col,...)
SELECT...;
```
Advantages of Multitable INSERTS

- Eliminates the need for multiple INSERT...SELECT statements to populate multiple tables
- Eliminates the need for a procedure to do multiple inserts using conditional logic
- Improves performance significantly over the above two methods due to the elimination of the cost of materialization and repeated scans on the source data
Applications of **MERGE** statements

- **MERGE** statements use a single SQL statement to complete an **UPDATE** or **INSERT** or both.
- The statement can be parallelized transparently.
- The array interface can be used for bulk operations.
- Performance is improved because fewer statements require fewer scans of the source tables.
Merge Syntax

**INTO Clause**
Use the INTO clause to specify the target table you are updating or inserting into.

**USING Clause**
Use the USING clause to specify the source of the data to be updated or inserted. The source can be a table, view, or the result of a subquery.

**ON Clause**
Use the ON clause to specify the condition upon which the MERGE operation either updates or inserts. For each row in the target table for which the search condition is true, Oracle updates the row based with corresponding data from the source table. If the condition is not true for any rows, Oracle inserts into the target table based on the corresponding source table row.

**WHEN MATCHED | NOT MATCHED**
Use these clauses to instruct Oracle how to respond to the results of the join_condition. You can specify these two clauses in either order.
Example of Using the MERGE Statement in Data Warehousing

```sql
MERGE INTO customer C USING cust_src s
ON (c.customer_id = s.src_customer_id)
WHEN MATCHED THEN
    UPDATE SET c.cust_address = s.cust_address
WHEN NOT MATCHED THEN
    INSERT (customer_id, cust_first_name, ...)
    VALUES (src_customer_id, src_first_name, ...
...);
```

Example of **MERGE** Statement

This an example of using the MERGE statement in data warehousing. `customer` is a large fact table and `cust_src` is a smaller table with rows that must be inserted into `customer` conditionally. A table such as the `cust_src` table is sometimes said to contain the “deltas” or changes that in this case need to be applied to the other table.

This MERGE statement indicates that the `customer` table must be merged with the rows returned from the evaluation of the ON clause. The ON clause in this case is the `cust_src` table, but it can be an arbitrary query. Each row from the `cust_src` table is checked for a match to any row in the `customer` table by satisfying the join condition specified by the ON clause. If so, each row in the `customer` table is updated using the UPDATE SET clause of the MERGE statement. If no such row exists in the `customer` table, then the rows are inserted into the `customer` table.
Summary

In this lesson, you should have learned how to:
• Use the new WITH clause
• Simultaneously insert data into multiple tables
• Use the MERGE statement to conditionally update or insert data

Summary

In this lesson, you should have learned about some techniques used in data warehousing that help to transform, transport, and load data.